FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Atualização de Demonstrador Robótico para Utilização do "ROS"

Nuno Miguel Baptista dos Santos

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Orientador: Prof. Doutor Armando Jorge Miranda de Sousa

Junho de 2013

Resumo

O ROS (Robotic Operation System) foi desenvolvido para facilitar a integração das várias funcionalidades do robô (locomoção, visão, navegação, localização, sensorização) e disponibiliza várias bibliotecas, serviços e ferramentas para aplicações robóticas. Este projeto utiliza o demonstrador de condução autónoma desenvolvido para o Festival Nacional de Robótica, criado noutro projeto e apresentado no IJUP'11 (Autonomous Driving Demonstrator - CondDois).

O objetivo de estudo passa por realizar o porte do sistema de controlo e locomoção desenvolvidos em várias plataformas e aplicar a migração para a plataforma única, ROS.

A locomoção diferencial do robô, constituída por dois motores *brushless*, é comandada com recurso ao driver da *Advanced Motion Controls*. No sistema desenvolvido anteriormente, para controlo dos motores, o sinal de comando e hodometria é realizado entre a comunicação da driver e da plataforma *Arduino*. Neste projeto, desenvolveu-se uma comunicação de dados entre computador e o motor, via porta de série e implementada em ROS, que elimina a necessidade da placa eletrónica *Arduino* para o controlo de velocidade.

A localização relativa do robô é baseada pela hodometria dos impulsos do hall sensor do motor e poderá em trabalhos futuros, usar uma localização absoluta através de marcadores localizados usando o sistema visão em tempo real.

O sistema de controlo será desenvolvido em C++ IDE (Integrated Development Environment) com a ajuda da visualização do simulador 3D (rviz) de ROS. Este simulador é uma ferramenta para testes, enquanto que o *software* de controlo é responsável pelo planeamento de caminho e decisões de orientação do robô.

Com a conclusão deste estudo, pretende-se que o robô seja capaz de realizar comportamentos de demonstração da condução autónoma com funcionamento completo em ROS. Na migração para ROS, espera-se que as partilhas de dados e funcionalidades do robô realizem-se em menos tempo e com maior capacidade de processamento, ao passo que os sistemas criados no futuro, sejam fáceis e práticos de integra-los com o sistema desenvolvido.

Abstract

The ROS (Robotic Operation System) was developed to facilitate the integration of the robot functions (locomotion, vision, navigation, positioning, sensing) and provide several libraries, services and tools for robotic applications. This project uses the autonomous driving demonstrator presented in the National Festival of Robotics and in the IJUP'11 (Autonomous Driving Demonstrator – CondeDois).

The objective of the study is to perform the modification of the systems of control and locomotion developed across platforms and apply the migration to a single platform, ROS.

The differential locomotion of the robot consists in two brushless motors controlled by Advance Motion Controls drivers. In the system previously developed for controlling the motors, the command signal and odometry communication is performed between the driver and the Arduino platform. In this study, the motor data communication implemented in ROS is replaced by a serial port, which eliminates the need to use the Arduino board for the velocity control.

The relative location of the robot is based on the odometer pulses from the hall sensor of the motor and, in future work, may use an absolute location by using real time vision system markers.

The control system will be developed in C++ IDE (Integrated Development Environment) with help of the ROS 3D simulator (rviz). This simulator can be used as a test tool for debugging purposes, while the software will control path planning and decisions needed to guide the robot.

The conclusion of this study is intended that the robot is able to perform behaviors demonstration of autonomous driving with full ROS operation. In the migration to ROS, it is expected that data sharing and robot features perform in less time, with larger processing capacity and improved interaction with other future systems.

Agradecimentos

Mesmo considerando que a Dissertação presente tratar-se de um trabalho individual, a sua realização não seria possível sem o apoio inquestionável de várias pessoas. Como tal, gostaria de deixar por escrito os meus mais sinceros agradecimentos às pessoas que contribuíram para a realização desta Dissertação. Entre as quais:

Ao Prof. Doutor Armando Jorge Miranda de Sousa, meu orientador, com quem tive a honra e prazer de trabalhar neste período de tempo, pela experiência e motivação transmitida, e pela oportunidade de desenvolver este trabalho;

A todos os meus amigos, por todo o apoio dado diretamente e indiretamente na elaboração deste trabalho. Em especial para aqueles que me acompanharam ao longo dos últimos anos na minha vida de estudante;

Aos meus irmãos João e Tiago, pela ajuda prestada, boa disposição transmitida e amizade inigualável;

E por último, aos meus Pais pelo apoio incondicional, incentivos e por terem sempre acreditado em mim.

Nuno Santos

"A journey of a thousand miles begins with a single step."

Lao-tzu

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

AMC	*Advanced Motion Controls*
CPR	Contrário dos Ponteiros do Relógio
CRC	*Cyclic Redundancy Check*
DCA	Demonstrador de Condução Autónoma
DOF	*Degrees Of Freedom*
FEUP	Faculdade de Engenharia da Universidade do Porto
FNR	Festival Nacional de Robótica
GUI	*Graphical User Interface*
IDE	*Integrated Development Environment*
LED	*Light-Emitting Diode*
LSB	*Least Significant Bit*
MIEEC	Mestrado Integrado em Engenharia Eletrotécnica e de Computadores
PCL	*Point Cloud Library*
PID	*Proportional-Integral-Derivative*
RGB	*Red, Green e Blue*
ROS	*Robotics Operating System*
RPM	Rotações Por Minuto
PR	Ponteiros do Relógio
PWM	*Pulse-Width Modulation*
SLAM	*Simultaneous Localization and Mapping*
UMBmark	*University of Michigan Benchmark test*
URDF	*Unified Robot Description Format*
USB	*Universal Serial Bus*

Capítulo 1

Introdução

O desenvolvimento do suporte à programação na área da robótica tem tido, ao longo dos tempos, constantes avanços dentro da comunidade científica, com aplicação em várias áreas do ramo da automação.

Dada a necessidade de partilha e reutilização da informação numa única plataforma a nível científico, criaram-se sistemas operativos robóticos que procuram criar ambientes e condições para o aparecimento de oportunidades e exploração de novas temáticas nesta área.

O *Robotic Operation System* (ROS) é um software, com várias soluções para implementar em diferentes tipos de robôs. Idealmente, o ROS é um sistema operativo para a área da robótica industrial e científica, onde procura implementar uma *framework* de bibliotecas e ferramentas com o principal objetivo de ajudar no desenvolvimento de robôs autónomos.

Neste contexto esta dissertação procura implementar uma plataforma idêntica a um sistema operativo para a robótica, denominado ROS, num robô já criado para demonstração da condução autónoma. Pretende-se que o robô com a utilização do software ROS potencie a sua capacidade em novas áreas, utilizando métodos e soluções adequadas para uma satisfatória e eficiente utilização do demonstrador.

1.1 Enquadramento e Motivação

Na atualidade, cada vez mais se faz recurso à utilização da robótica, como um meio que permite a cooperação entre este equipamento tecnológico e o trabalho humano, de forma a substituir e ampliar as tarefas realizadas pelas pessoas.

A função do programador passa por criar aplicações capazes de realizar tais tarefas, tornando o mundo da robótica uma área em expansão e bastante apelativa para os vários sectores da humanidade.

Numa sociedade cada vez mais exigente e com maiores necessidades torna-se pertinente aprofundar e atingir níveis superiores na área da robótica. Neste sentido, procura-se desenvolver uma plataforma universal, onde todo o conhecimento possa convergir de forma a obter-se melhores e eficientes resultados. O aparecimento de uma plataforma deste tipo, comprometido ao código

aberto e que se comporta como um sistema operativo para a robótica, leva a que a nível mundial vários programadores partilhem os seus conhecimentos e experiências.

O ROS, plataforma que se enquadra neste tipo de definição veio abrir inúmeras soluções, já que os vários modelos de robôs passam a integrar recursos e ferramentas para a realização de diversas tarefas.

Assim a utilização de robôs nas tarefas consideradas perigosas, difíceis, desgastantes e inacessíveis para o ser humano, é um tema que desperta interesse a nível científico, industrial e da sociedade em geral.

1.2 Objetivos

O objetivo principal deste projeto é implementar num robô, já previamente construído, funções de um demonstrador de condução autónoma em espaços estruturados e fechados, com recurso ao software ROS. Pretende-se assim aprofundar a plataforma ROS, melhorando as funcionalidades do robô nas suas várias vertentes.

Para tal será necessário um estudo sobre: as arquiteturas existentes na área da robótica autónoma, a plataforma ROS - aprofundando potenciais soluções para implementação no demonstrador de condução autónoma - e identificação e seleção dos casos de interesse, que serão implementados nas várias ferramentas do robô. Com este estudo pretende-se abordar parâmetros como a localização espacial, algoritmos de trajetórias e melhoramento ao nível das comunicações internas no funcionamento do demonstrador.

1.3 Estrutura do documento

No primeiro capítulo é realizado o enquadramento teórico da dissertação e são apresentados os objetivos, motivação, metodologia adotada e a estrutura do presente trabalho.

No Capítulo 2 é efetuado o estado de arte sobre os seguintes assuntos: Sistemas de Arquitetura Robóticas, Ferramentas e Funcionamento do ROS. Também neste capítulo é feito o levantamento de casos de estudo.

No Capítulo 3 é realizada uma introdução do projeto "CondeDois", estado inicial do demonstrador. Descreve-se, igualmente, a implementação dos diferentes parâmetros que são alvo de estudo: seguimento de trajetórias, comunicação com driver dos motores, controlo com base na posição e os sistemas desenvolvidos em ROS. Por último esclarece-se sobre o desenvolvimento do sistema de LEDs, incorporados no demonstrador e respetivo protocolo de comunicação.

No Capítulo 4 são apresentados os resultados obtidos dos ensaios e simulações efetuadas no capítulo anterior, concretamente, o comportamento dos motores e comportamento do demonstrador.

Por último, no Capítulo 5 são apresentadas as conclusões do trabalho realizado e as perspetivas e sugestões para a continuação dos estudos realizados na presente Dissertação.

Capítulo 2

Estado da Arte

2.1 Sistemas de Arquiteturas Robóticas

Existem, atualmente, várias plataformas de ajuda no desenvolvimento de aplicações robóticas, entre as quais a *Robotic Operating System* (ROS), *Urbi*, *Microsoft Robotics Studio*, *Dave's Robotic Operating System* (DROS). (figura 2.1)

A maioria dos softwares em código aberto, com exceção da *Microsoft Robotics Studio* que necessita de licença comercial para utilização, permitem uma partilha de informação e de aplicações desenvolvidas por membros da comunidade robótica.

Figura 2.1: Sistemas de Arquiteturas Robóticas [1] [2] [3] [4]

Estes softwares possuem ferramentas e conteúdos API (*Application Programming Interface*) que permitem utilizar as suas funcionalidades sem a necessidade de desenvolver detalhes da implementação destes.

O destaque do ROS em relação às restantes arquiteturas fica-se a dever à implementação de um sistema de comunicação entre as funcionalidades robóticas, pois permite que a troca de diversos tipos de mensagem seja rápida, *multithread* e fácil de implementar. Esta plataforma de código aberto, sob licença BSD, e que tem respondido às necessidades da área robótica, tem sido alvo de intensa investigação, permitindo a criação e desenvolvimento de novas aplicações robóticas, aumentando assim a visibilidade do ROS (figura 2.2).

Estas novas especificidades do ROS levaram a que este fosse integrado e adotado por outros softwares.

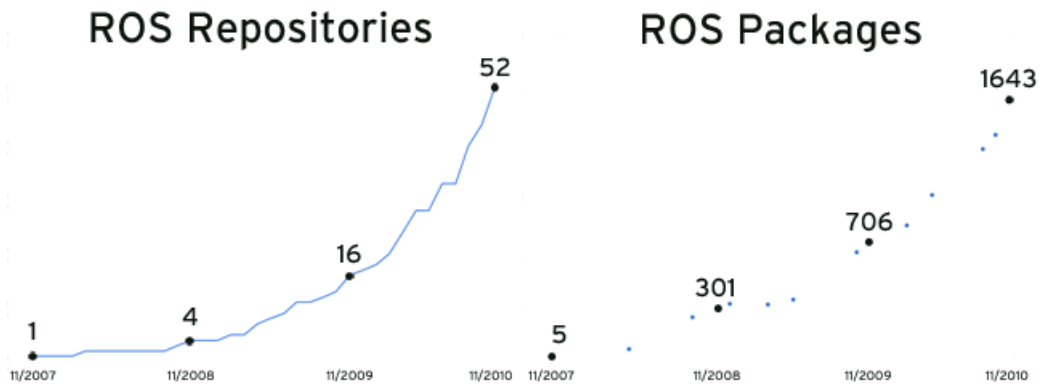


Figura 2.2: Evolução dos repositórios e *packages* do ROS (2007-2010) [1]

A título de exemplo, a *Urbi* desenvolveu no seu software a capacidade de utilizar comandos existentes no ROS. Com o mesmo objetivo, a *Orocos*, um software de controlo, implementou uma integração do ROS de forma a que o seu software aceite pacotes provenientes desta plataforma [13].

2.2 Robotic Operation System

Robotic Operation System (ROS) é um software, desenvolvido em 2007, com várias soluções robóticas para implementar em qualquer tipo de robô (figura 2.3). Idealmente, é um sistema operativo para a área da robótica, onde o principal objetivo é a criação de uma plataforma global para o desenvolvimento e partilhas de trabalhos ligados à robótica.

Figura 2.3: Logótipo do ROS [1]

Esta plataforma de código aberto, inicialmente desenvolvido pelo Laboratório de Inteligência Artificial de Stanford e atualmente pela *Willow Garage*, utiliza bibliotecas e ferramentas para integrar e reutilizar conjuntos de códigos desenvolvidos nesta plataforma.

O ROS disponibiliza ferramentas e reutiliza bibliotecas de outros projetos de código aberto, aproveitando o trabalho de aplicação robótica nas áreas de navegação, simulação, visão, perceção e controlo, sendo exemplo as bibliotecas de processamento de imagem (opencv, pcl) e simuladores (stage, gazebo) [14].

Foram lançadas várias versões do ROS, sendo algumas incompatíveis entre si e genericamente referenciadas pelo código nome, em vez do tradicional número de versão. Atualmente o ROS encontra-se na versão Groovy Galapagos.

- Box Turtle (Março, 2010): primeira distribuição do ROS, com 6 *stacks* iniciais do software para robôs genéricos e PR2 (figura 2.4);

Figura 2.4: Robô *PR2* [1]

- C Turtle (Agosto, 2010): segunda distribuição do ROS, com a principal mudança, o crescimento para 20 *stacks*;
- Diamondback (Março, 2011): na terceira distribuição foi incluído *stacks* para utilização do *Kinect* (OpenNI) e PCL (*Point Cloud Library*);
- Electric (Agosto, 2011): a quarta distribuição do ROS, inclui um aumento e atualização das *stacks*, à semelhança dos casos do openCV e do PCL;
- Fuerte (Abril, 2012): a quinta distribuição significativa do ROS, inclui a integração com outros frameworks softwares e ferramentas como o caso Qt. Instalação do opencv passa a ser opcional para uso no ROS;
- Groovy Galapagos (Dezembro, 2012): a sexta e atual distribuição do ROS, aumenta o número de arquiteturas, sistemas operativos, hardware e robôs disponíveis para utilização do ROS. Desaparecimento do conceito *stacks* e concentração do sistema de controlo de versões para *github* são as principais alterações.

2.2.1 Ferramentas do ROS

O ROS oferece um leque de ferramentas e funcionalidades na área da robótica, assim como um software onde a implementação passa por utilizar os seus serviços. As funcionalidades do ROS são apresentadas na tabela 2.1 e as ferramentas deste são descritas posteriormente.

É de salientar que, a plataforma ROS providencia ferramentas exclusivas para alguns tipos de linguagem de programação, sendo C++ e *Python* as duas linguagens principais e com maior número de pacotes disponíveis. O ROS suporta ainda outras linguagens como *LISP* e *Octave*.

Existem diversas ferramentas disponibilizadas no ROS, sendo a rviz, rxtools e as dos comandos do terminal as mais usuais.

Tabela 2.1: Funcionalidades do ROS [11]

Funcionalidades	ROS (serviços e tópicos)	Linguagem C++	Linguagem Python
ROS (system)	ROS	roscpp	rospy
Tipo de dados básicos	common_msgs	common_msgs	common_msgs
Manipulação de mensagens	topic_tools	message_filters	message_filters
Drivers	joystick_drivers, camera_drivers, laser_drivers, sound_drivers, imu_drivers	joystick_drives, camera_drives, laser_drivers, sound_drivers, imu_drivers	
Implementação de drivers	driver_common	driver_common	
Filtros de dados		filters	
Processamento 3D	laser_pipeline, point_cloud_perception	laser_pipeline, point_cloud_perception	
Processamento de imagem		image_common, image_pipeline, vision_opencv	vision_opencv
Tranformações/Coordenadas		tf, tf_conversions, robot_state_publisher	tf, tf_conversions
Ações	actionlib	actionlib	actionlib
Manutenção de tarefas	executive_smach		executive_smach
Navegação	navigation		
Simulação (2D)	simulator_stage	simulator_stage	
Simulação (3D)	simulator_gazebo	simulator_gazebo	
Modelo Robô		robot_model	
Controladores tempo real	pr2_controller_manager	pr2_controller_manager, realtime_tools	
Planeamento de movimentações (Manipuladores)	ompl, chomp_motion_planner, sbpl	actionlib, move_arm	actionlib, move_arm

Rviz é um visualizador de ambientes, onde é possível combinar modelos de robô, dados de sensores como o *kinect* ou como laser *Hokuyo*, ambiente gráfico em 3D permitindo a combinação de diversos ângulos de vista (figura 2.5).

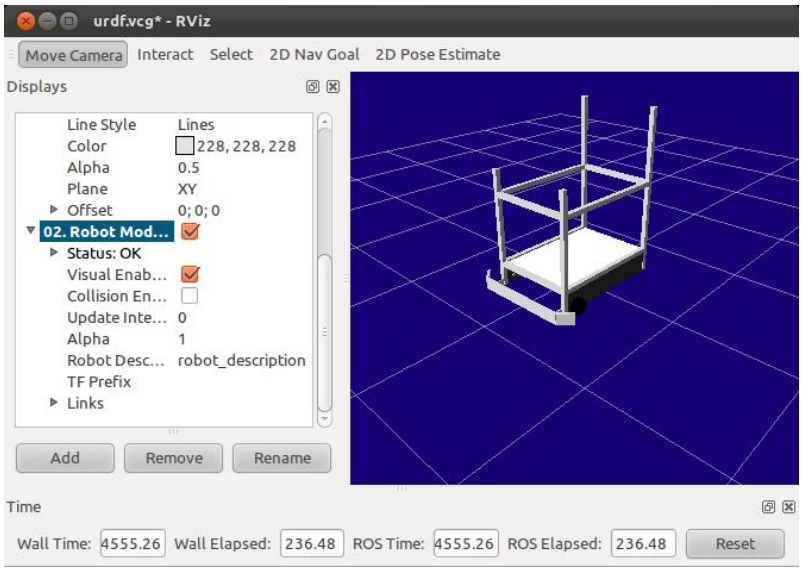


Figura 2.5: Ambiente 3D da ferramenta rviz

Rxtools é um conjunto de ferramentas (rxgraph, rxbag, rxplot) que permitem a visualização e analise dos *nodes* do sistema. Rxgraph permite mostrar a visualização de *nodes*, *topics* e informação das mensagens em todo o sistema. Rxbag permite fazer registo do histórico de operações e

dados dos *nodes*, de forma a poderem ser analisados posteriormente (figura 2.6).

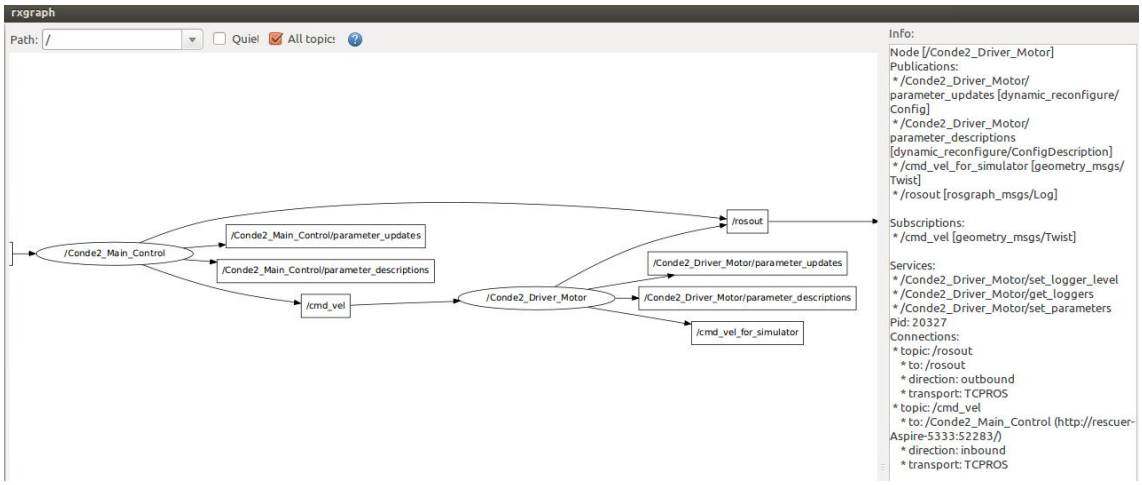


Figura 2.6: Visualização de um sistema ROS, com a ferramenta rxgraph

A maior parte das ferramentas, serviços utilizados no ROS, são na linha de comandos (consola). A tabela 2.2, mostra os principais comandos e descrição das suas funções.

Tabela 2.2: Ferramentas da linha de comandos do ROS

Comando	Descrição
rospack	fornece informação relativa aos *packages* do ROS;
rosstack	fornece informação relativa aos *stacks* do ROS;
roscd	desloca cursor da consola para localização da *stack* ou *package*;
rosls	lista ficheiros do *package*;
roscp	copia ficheiros do *package* de/para;
rosmsg	fornece informações relativa às definições das mensagens no ROS;
rossrv	providencia informações relacionadas com os serviços ativos do ROS;
rosmake	compila *package* do ROS;
roscreate-pkg	cria um novo *package*;
roscreate-qt-pkg	cria um novo *package* ROS+Qt;
rostopic	ferramenta que mostra informação acerca do *topics* ativos no ROS;
rosrun	permite correr o executável do *package*;
roscore	inicia o servidor principal do ROS;
roslaunch	ferramenta de lançamento de múltiplos *nodes* localmente;
rosnode	mostra informações em tempo real do node;
rosdep	instalação de dependências do ROS;

A organização do ROS, tem como base principal o funcionamento em modos de pacotes (em inglês denominado por *package*) onde são inseridos a documentação, informação e ficheiros ligados à parte de programação. A criação destes pacotes são automáticas, originando que a partilha dos mesmos seja de uma forma genérica no que toca a estrutura dos ficheiros.

Assim a criação de pacotes tem a seguinte estrutura:

- bin (diretório) : repositório dos ficheiros executáveis.
- build (diretório) : ficheiros criados da complicação cmake.
- lib (diretório) : repositório das bibliotecas.
- include (diretório) : diretório para ficheiros adicionais de imagens, recursos da aplicação.
- src (diretório) : repositório dos ficheiros de programação cpp, python.
- manifest.xml : providencia informações sobre dependências do pacotes, autor, licença e descrição.
- CMakeList.txt : ficheiro que contém os comandos necessários para a construção do executável.

No anexo D mostramos como se realiza a migração de uma aplicação *Cmake* em *C++* para a criação de um pacote no *CmakeList* do ROS.

2.2.2 Funcionamento do ROS

A comunicação do ROS baseada em eventos, utiliza processos para publicar/subscrever nós a tópicos. Concetualiza-se que os nós publicam e subscrevem os tópicos, permitindo assim uma comunicação entre nós, onde recebem mensagens, apenas os que estiverem subscritos ao tópico.

A figura 2.7 demonstra uma simples publicação e subscrição de dois nós a um tópico. Desta forma sempre que o nó publicar novas mensagens os nós que estejam subscritos ao tópico irão receber a informação.

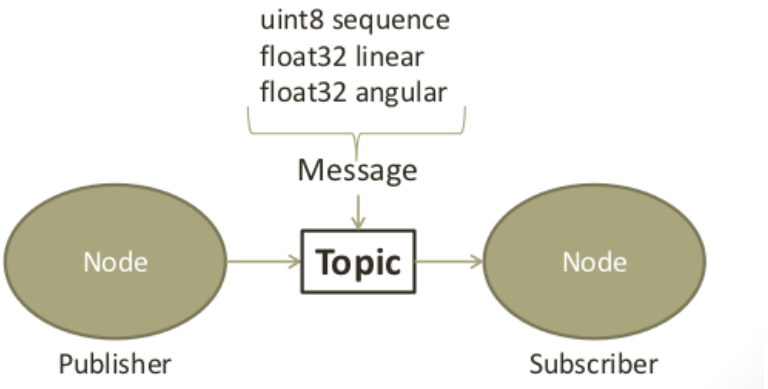


Figura 2.7: Sistema publicar/subscrever ROS

Os nós correspondem à execução de um programa e são criados no sistema, geralmente, via consola e sempre com o servidor mestre do ROS (*roscore*) iniciado. O servidor *roscore* é responsável por inicializar várias dependências, bibliotecas do ROS e o sistema de comunicação entre nós.

Nos tópicos do sistema, a partilha de mensagens é realizada em programação computacional por *callbacks*, permitindo assim uma comunicação entre processos e *threads* do sistema operativo. O tipo de mensagens existentes no ROS é bastante alargado, sendo sempre necessário converter os dados para formatos compatíveis. Por exemplo, a partilha de mensagens entre nós no formato de imagem do *opencv* (*IplImage*) necessita de conversão para tipo *sensor_msgs::Image* fornecido pelo pacote *cvBrigde* do ROS.

A diversidade no tipo de mensagens, facilidade e segurança na comunicação, partilha de dados, como a hodometria, navegação ou imagem são os principais motivos do aumento exponencial do uso da plataforma do ROS na comunidade científica.

2.2.3 Tipo de mensagens do ROS

Qualquer informação proveniente da arquitetura de software externa à plataforma do ROS, requer que seja transformada e convertida para o tipo de formato das mensagens da plataforma.

Como foi indicado anteriormente o ROS possuí grande diversidade no tipo de mensagens, o que facilita a conversão dos dados para formatos do tipo de mensagem compatível nesta plataforma. Muitos destes tipos de mensagens foram elaborados para aplicações robóticas, permitindo a integração de diversas áreas, como navegação, geometria e visão.

Na tabela 2.3 são apresentadas os principais tipos de mensagens fornecidas pela plataforma ROS.

Tabela 2.3: Formatos dos tipo de mensagens do ROS

Package	Tipo de mensagem	Orientadas às áreas
std_msgs	*string, bool, int8, int16, char, float, MultiArrays*	Mensagens no formato comum da programação;
geometry_msgs	*twist, transform, pose, point*	Geometria primitiva como pontos, vetores, posições ou velocidades;
nav_msgs	*odometry, path, occupancyGrid*	Mensagens indicadas para a navegação;
trajectory_msgs	JointTrajectory	Orientação para junção de trajetórias;
visualization_msgs	ImageMarker, InteractiveMarker	Utilizadas na camada de alto nível, como a ferramenta *rviz*;
stereo_msgs	DisparityImage	Especificas para processamento *stereo*, como disparidade de imagens;
sensor_msgs	Image, PointCloud, LaserScan	Definidas para usar em sensores, como câmaras, laser scan, infravermelhos;

2.3 Caso de estudo do ROS

Cada vez mais a utilização do ROS é aplicada em diversos robôs, como os projetos *Care-O-bot*, *iRobot Create*, *Aldebaran Nao* ou PR2 apresentados nas páginas seguintes.

A figura 2.8 mostra o crescimento dos robôs com rotinas já implementadas no ROS dando uma parte de suporte a estes.

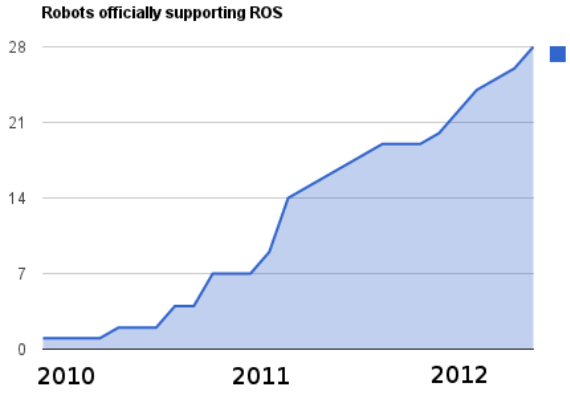


Figura 2.8: Evolução dos robôs que, oficialmente, suportam ROS [1]

iRobot Create

A corporação *iRobot*, fundada em 1990, cria e desenha robôs capazes de realizar diversos tipos de tarefas. A construção do *iRobot Create*, um robô com as especialidades de associar diferentes tipos de sensores, de reprogramar tarefas e com baixo custo, torna esta plataforma robótica móvel uma ferramenta interessante para fins comerciais [5].

Figura 2.9: Robô *Roomba* da *iRobot* [5]

O *iRobot Create* é uma versão do robô *Roomba* (figura 2.9), sendo utilizado, preferencialmente, como uma plataforma para experimentações e demonstrações. Este é apenas vendido nos Estados Unidos da América, tendo um acesso restrito a este espaço, enquanto que o *Roomba* é comercializado em todo o mundo, permitindo um uso mais alargado.

Várias comunidades científicas criam e partilham os seus desenvolvimentos nos repositórios do ROS, como o caso da *Brown University* que disponibiliza drivers para ligação do ROS à câmara instalada no robô *iRobot Create* (figura 2.10a). Utilizando a mesma plataforma de hardware, a *Willow Garage* mostra o potencial do *iRobot Create* fazendo recurso ao sensor *Kinect* (figura 2.10b) [1].

(a) Exemplo de uma aplicação no robô *iRobot Create*

(b) Robô *iRobot Create* com o sensor *Kinect*

Figura 2.10: Projetos iRobot com utilização do ROS [1]

Apesar do *iRobot Create* ser vendido a baixo custo, e tento em mente a limitação do seu uso, o ROS disponibiliza pacotes para o robô *Roomba*, que permitem a configuração e manipulação ao nível dos sistemas de navegação e de simulação.

Toda a informação fornecida pelo ROS, como os exemplos acima referidos, podem ser descarregada diretamente dos seus repositórios ficando disponível para a sua utilização.

Com estas soluções de *software* partilhadas nos repositórios do ROS, permitem ao investigador o acesso ao seu conteúdo, a sua alteração e a sua reutilização para outras tarefas na plataforma [15].

Care-O-bot 3

Care-O-bot 3 é um manipulador da *Fraunhofer IPA*, desenvolvido na Alemanha (figura 2.11). O robô é projetado para ter aparência de um ser humano, com uma bandeja e um braço leve *Schunk*, que pode recolher objetos e colocá-los na bandeja. O software do robô a utilizar na plataforma ROS, assim como toda a documentação de instalação, configuração e tutoriais estão disponíveis no site oficial do ROS [1].

No repositório do *Care-O-bot 3* encontram-se vários pacotes entre os quais os mais importantes são:

- Modelo robô em formato URDF (*Unified Robot Description Format*) - Formato URDF que contém a informação do robô, no aspeto visual e estrutural da apresentação em simuladores. Permite utilizar este modelo 3D do robô em vários simuladores, que suportam este formato;
- Ligação aos sensores e drivers;
- Simulador 3D do robô;
- Ação e cálculo de movimentos do braço manipulador;
- Navegação e controlo remoto do *Care-O-bot 3*.

Figura 2.11: Robô *Care-O-bot 3* utiliza tecnologia ROS [1]

Aldebaran Nao

Aldebaran Nao é um robô humanoide, desenvolvido pela *Aldebaran Robotics* com finalidades educacionais (figura 2.12). O custo moderado, diversidade dos sensores, possíveis soluções e utilizações das funções, torna este robô um dispositivo de excelência para a aprendizagem.

O robô humanoide integra uma camada sensorial, incluindo duas câmaras, quatro microfones, sensor sonar *rangefinder*, emissor e recetor de infravermelhos, sensores de tato e de pressão. Têm também associado, 25 DOF (graus de liberdade), sincronização de voz e bateria LiPO (*Lithium-ion polymer batteries*) [15][6].

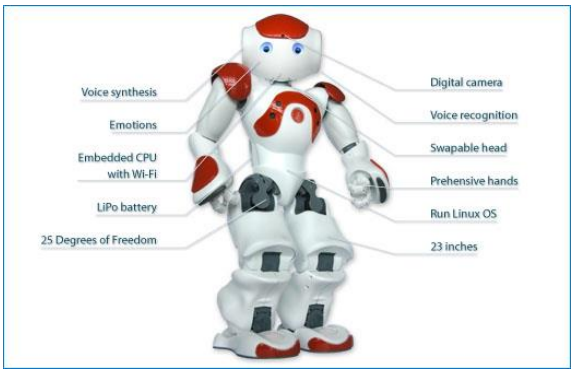


Figura 2.12: Robô *Albedaran Nao* [6]

Aplicações ROS disponíveis para o robô *Nao*:

- Instalação, configuração e modelo robô em formato URDF;
- Ligação aos sensores e drivers;
- Utilizar Simulador *NAO* e conectar localmente ao robô;
- Ação e cálculo de movimentos das pernas do robô;
- Navegação e controlo remoto do *Albedaran Nao*.

Willow Garage PR2

Projetado pela *Willow Garage*, o robô PR2 é um dos ícones do ROS. Esta entidade desenvolve, atualmente, a plataforma ROS e criou em 2007 o PR2 que corre unicamente este sistema de software.

PR2 é um manipulador móvel virado para a área da investigação, onde diversos sensores como *Kinect*, *Hokuyo* e as especificações de manipulação do robô, fazem o PR2 uma plataforma de desenvolvimento com várias potencialidades na robótica (figura 2.13).

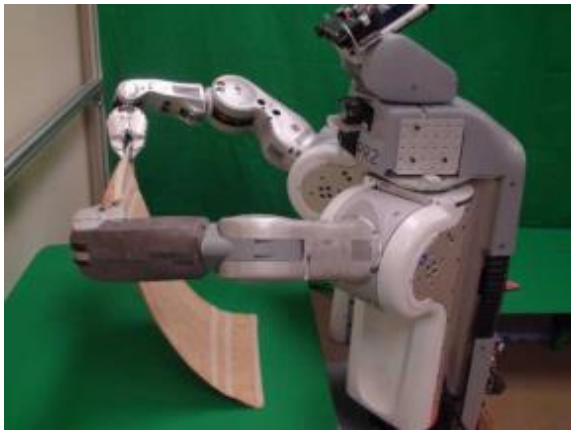


Figura 2.13: Robô PR2 desenvolvido pela *Willow Garage* [1]

Sendo o robô com mais conteúdos no repositório do ROS, podemos encontrar pacotes, que englobam vários parâmetros, tais como: controlo remoto, sistema de navegação, sistema de visão, cinemática, manipulação do robô (ex.: braço robótico), execução de tarefas, simulação do modelo do robô e drivers para motores e sensores (laser, câmaras, som).

Capítulo 3

Atualização do Demonstrador para utilização ROS

Neste capítulo apresenta-se o estado inicial do robô e estudos desenvolvidos ao longo do projeto na plataforma do ROS.

Na secção 3.1 é apresentado o projeto do Demonstrador de Condução Autónoma, projeto este desenvolvido noutra dissertação. Na secção 3.2 são expostos os processos portados para a plataforma ROS. Na secção 3.3 descreve-se o funcionamento do motores e protocolo de comunicação. Na secção 3.4 relata-se o sistema de controlo do Demonstrador de Condução Autónoma. Na última secção do capítulo (3.5) faz-se referência a alguns sistemas desenvolvidos ao longo do projeto da dissertação.

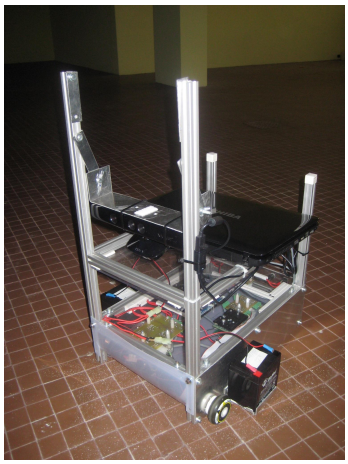
3.1 Demonstrador de Condução Autónoma

O Demonstrador de Condução Autónoma (DCA) desenvolvido durante a realização do projeto ″Demonstrador de Condução Autónoma″ tem como objetivo o desenvolvimento de um robô para fins demonstrativos e destinado ao público em geral, ou seja, menos especializado [7].

A figura 3.1 representa, respetivamente, o estado anterior (fig. 3.1a) e o estado atual (fig. 3.1b) do Demonstrador de Condução Autónoma.

No âmbito do projeto, o conjunto de requisitos do demonstrador carateriza-se por: [7]

- Plataforma móvel;
- Custo reduzido;
- Utilização simples;
- Atuação tanto em ambientes reduzidos como em ampliados;
- Sem limitação a ambientes estruturados;
- Possuir um sistema de deteção e identificação de objetos;
- Efetuar manobras atrativas para o público menos especializado, como efetuar dois percursos pré-definidos, um do tipo *boomerang* e outro circular em torno de um objeto e um percurso gerado de forma inteligente, com desvio de um obstáculo, à *priori*, detetado.



(a) Estado inicial do DCA [7]

(b) Estado atual do DCA

Figura 3.1: Estrutura do Demonstrador de Condução Autónoma

Nesta dissertação, estuda-se e desenvolve-se os sistemas de locomoção e controlo, aplicados no ROS, tendo-se mantido todos os anteriores requisitos do outro projeto, exceto as vertentes ligadas ao sistema de visão.

3.1.1 Arquitetura do CondeDois

CondeDois é a designação atribuída ao demonstrador de condução autónoma, sendo este um robô de médias dimensões e com locomoção diferencial. A arquitetura do robô está dividida em três camadas, nomeadamente, uma de baixo nível, outra de nível intermédio (camada de interface) e outra de alto nível.

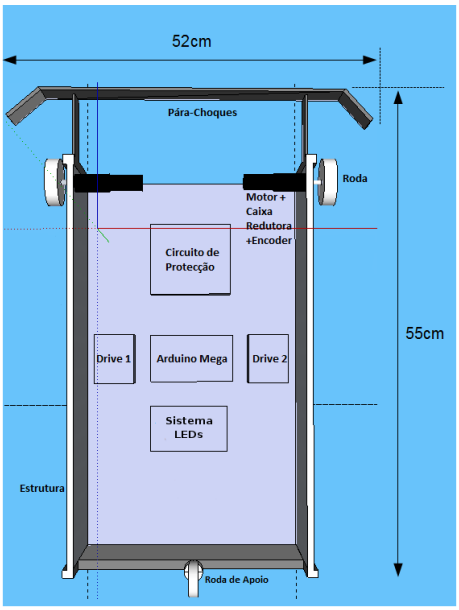


Figura 3.2: Perspetiva da estrutura por debaixo do robô [7]

Na figura 3.2 representa a perspetiva de baixo da locomoção diferencial do robô, com duas rodas motriz e uma roda castor (roda livre traseira).

A ligação entre os componentes das diversas camadas da arquitetura de *harware* inicial do projeto, pode ser visualizada na figura 3.3.

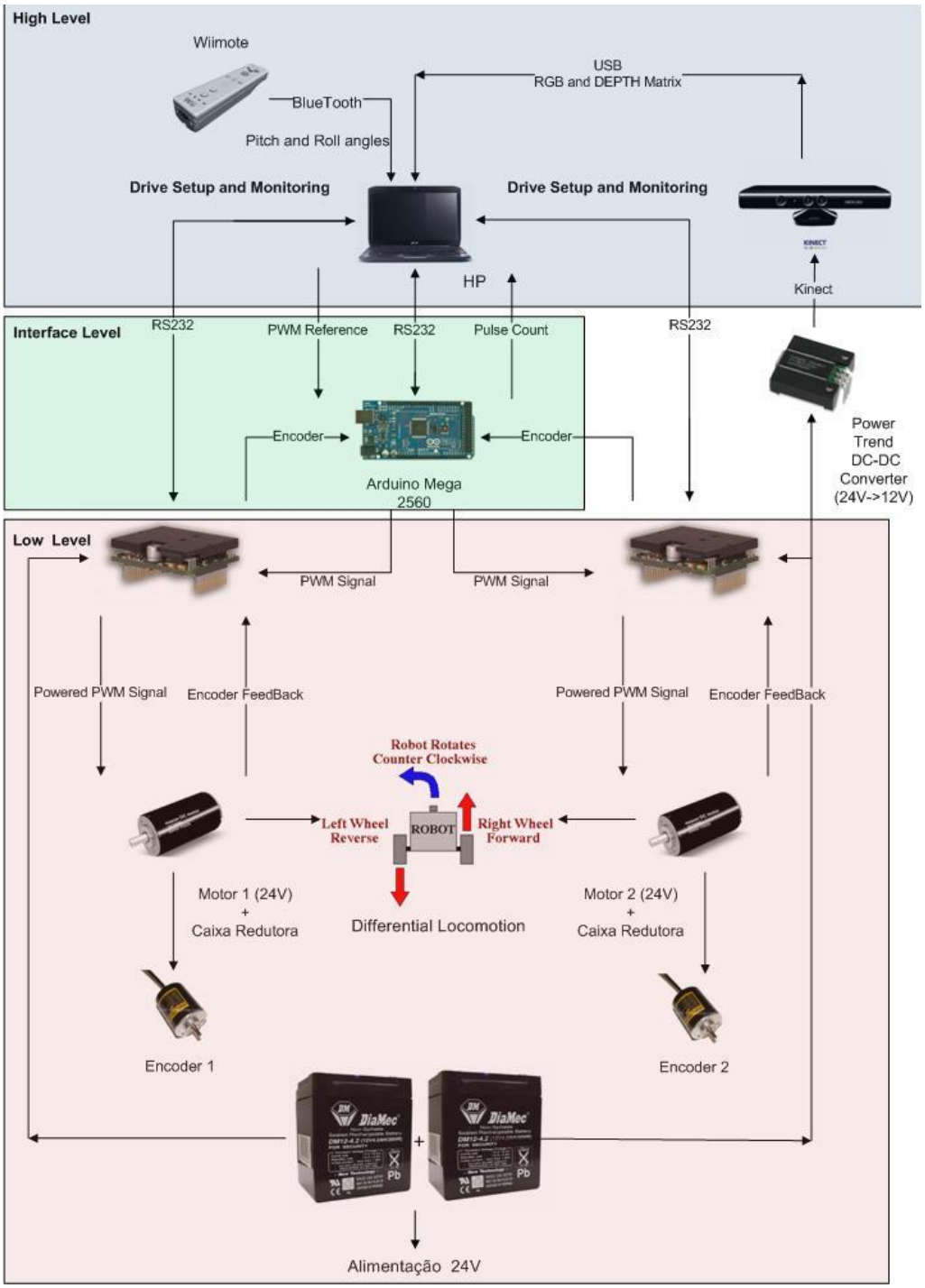


Figura 3.3: Arquitetura inicial do *harware* [7]

Com base do que se pretendia portar para a plataforma ROS, foi necessário alterar diversos

componentes entre as várias camadas, tornando o sistema mais seguro e estável, dado a interface entre as camadas, não ficar apenas dependente de um único componente. A arquitetura desenvolvida neste projeto mantém as três camadas lógicas, sendo a sua principal alteração a comunicação direta entre o computador e a *driver* de controlo dos motores.

No controlo de todo o sistema encontra-se o computador, na camada de alto nível. Esta camada executa todas as operações do *software* desenvolvidas na plataforma ROS, desde a recolha da informação proveniente dos sensores *hall* ao comando de velocidade para os motores.

A camada de interface (nível intermédio) é o módulo que permite a ligação do sistema de alto nível com o sistema de baixo nível e onde se encontra o dispositivo *Arduino Mega 2560* e as *drivers* da *Advanced Motion Controls* (AMC). A função do componente *Arduino* é de detetar e receber os sinais lógicos do sensor *hall*, enquanto que a *driver* encarrega-se do envio de comandos de velocidade e da comunicação com os motores.

A camada de baixo nível é composta por dois motores *brushless* 50 watts, com sensores *hall* acoplados a cada um, por um sistema de proteção para cada motor, com o recurso a relés 24V DC, por duas drivers da *Advanced Motion Controls* que realizam a locomoção do robô e por um sistema de *light-emitting diode* (LED). Todo o sistema de baixo nível é alimentado através de duas baterias de chumbo 12V/4.0Ah, denominado por camada de alimentação.

A ligação entre os elementos das diversas camadas pode ser visualizada pela arquitetura do hardware do demonstrador (fig. 3.4).

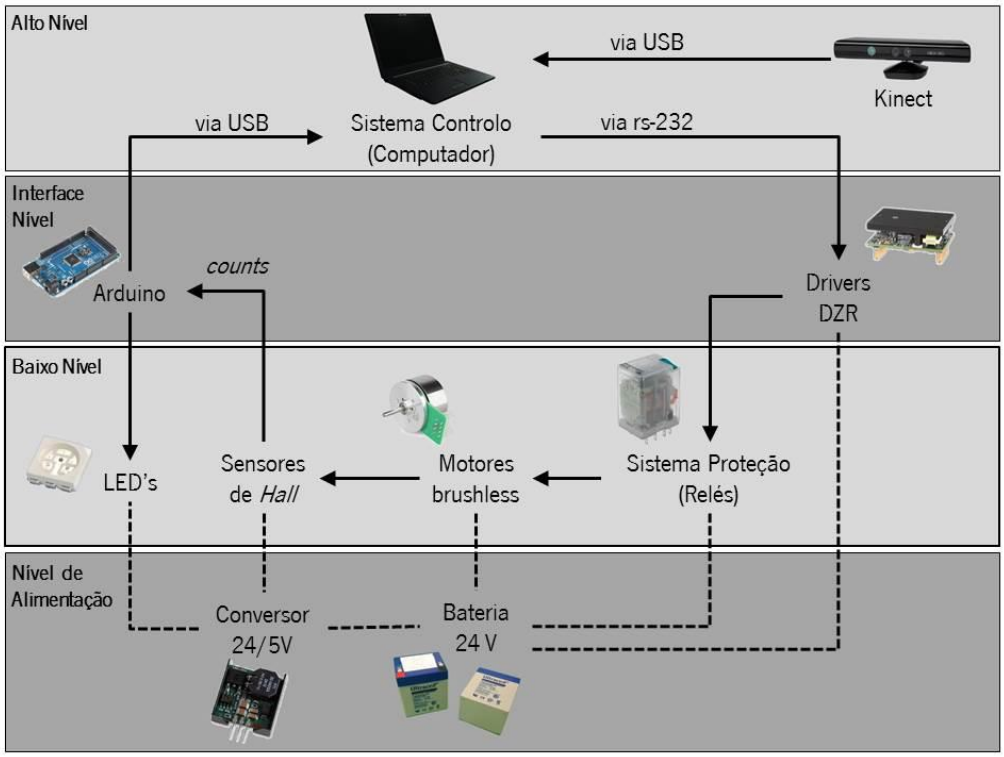


Figura 3.4: Arquitetura atual do *hardware*

3.2 Porte da plataforma

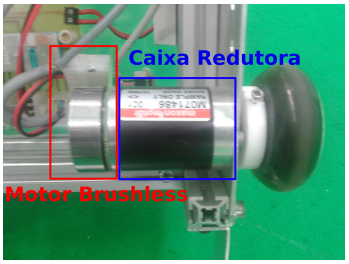
Nesta dissertação desenvolve-se a migração do sistema de locomoção e controlo para a plataforma ROS. No sistema de locomoção foi desenvolvido um *package* de configuração e comunicação, via porta USB/RS232, entre o computador e as drivers *Advanced Motion Controls*. Este sistema é responsável pela locomoção do robô, fazendo a ponte de ligação entre a camada de alto nível (computador) com a camada de baixo nível, drivers, que por sua vez irão controlar os motores.

No sistema de controlo, criou-se um *package* responsável pela decisão da navegação do demonstrador, decidindo a melhor trajetória com base nas informações recolhidas da hodometria. A informação da posição relativa do robô é recolhida através dos sensores de *hall* acoplados aos motores e processada já na plataforma ROS desde o *Arduino*, responsável pelo envio da deteção dos impulsos elétricos dos sensores de *hall*, até à informação gráfica no simulador realizada ao alto nível da arquitetura do demonstrador.

3.2.1 Estrutura do DCA

Na estrutura do robô a maior alteração foi a substituição dos motores DC para motores *brushless* - maxon EC 45 flat de 50 Watts (figura 3.5) integrados com sensores de *hall* para *feedback* da posição. Na implementação dos novos motores optou-se por alterar a comunicação pc->driver via rs-232 substituindo assim a antiga comunicação pc->arduino->driver que permitiu libertar espaço para adicionamento de novas tarefas ao *Arduino Mega 2560* como o caso do sistema de controlo dos LEDs. Apesar da troca de motores possibilitar uma otimização da organização da estrutura do robô, com menos ligações entre os componentes, acarreta a desvantagem da *driver dzr* não está equiparada e preparada para aceitar sensores de *hall* para *feedback* da posição pela via rs-232, devido à pouca precisão que englobam estes sensores. A alternativa encontrada para a contagem dos impulsos elétricos provenientes dos mesmos, é que seja realizada via arduino->computador.

(a) Motor *Brushless* com sensores de *Hall* [16]

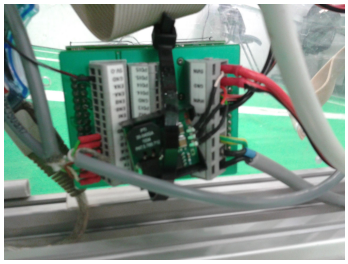


(b) Montagem dos motores

Figura 3.5: Motores implementados no robô

Com esta alteração a comunicação via RS232 não é apenas para configuração das drivers, mas também passa pelo envio de comandos para os dois motores melhorando os seus tempos, já que

não há necessidade da ligação no modo de ponte do *Arduino* como o seu envio do pulse-width modulation (PWM) para controlo da velocidade. (figura 3.6)



(a) Driver DZR da *Advanced Motion Controls*

(b) *Arduino Mega 2560* [8]

Figura 3.6: Montagem dos componentes *Arduino* e *drivers*

3.3 Sistema de Locomoção

O sistema de locomoção é constituído por dois motores *flat EC45* de 50 *watt* com sensores de *hall* acoplados, duas *drivers* de controlo *dzralte-012L080* da *Advance Motion Controls* e duas rodas.

O Modo de comando das drivers será por velocidade em malha fechada, obtendo um feedback das rotações dos motores por sensores de *hall*.

Motores

O motor disponível foi desenvolvido pela empresa *Maxon Motors*. Este componente pertence à série *EC 45 flat* e tem potência nominal de 50 *Watt*. As características do motor *brushless* encontram-se na tabela 3.1.

Tabela 3.1: Características do motor *EC 45 flat*

Motor Brushless	Características
Modelo	EC 45 flat
Tipo	Brushless
Potência nominal	50 W
Alimentação	24 V
Velocidade nominal	5260 RPM
Torque nominal	84.3 mNm
Corrente nominal	2.36 A
Corrente em repouso	201 mA
Corrente de arranque	24.5 A
Resistência térmica	0.978 Ω
Indutância térmica	0.573 mH
Eficiência máxima	83%

3.3.1 *Arduino Mega*

O *Arduino Mega* é responsável pela leitura dos movimentos dos motores, através de interrupções externas ocorridas nos sensores de hall. A deteção dos impulsos é feita através da subida da alteração do nível lógico de cada sensor. Isto permite através do conhecimento do diagrama de sequências dos impulsos elétricos provocados na comutação eletrónica (figura 3.7) saber a cada momento o posicionamento, sentido e velocidade de cada roda.

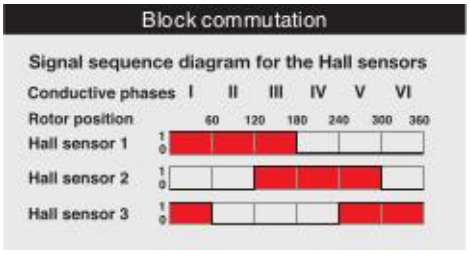


Figura 3.7: Sequência dos sensores *hall* [8]

Optou-se pela placa micro-controlador *Arduino Mega* pois é um dispositivo de fácil integração na estrutura do robô, devido às suas pequenas dimensões, de baixo custo para as funcionalidades e potencialidades apresentadas e que permite leituras rápidas dos impulsos elétricos provenientes dos sensores de *hall*, a frequência de relógio do micro-controlador de 16MHz. Outra vantagem do *Arduino Mega* são os seis pinos de saídas de interrupções externas, detetando assim todas as sequências dos sensores de *hall*. As características da placa *Arduino* são apresentadas na tabela 3.2 [8].

Tabela 3.2: Características do *Arduino Mega 2560*

Descrição	Características
Micro-controlador	ATmega2560
Alimentação	USB ou conversor AC-DC
Tensão de alimentação	5V
Corrente DC admissível por pino	40mA
Corrente DC máxima admissível	200mA
Nº pinos de Entradas/Saídas digitais	54
Nº pinos de Saídas PWM	15
Nº pinos de Entradas analógicas	16
Nº pinos com interrupções externas	6
Frequência de relógio	16 MHz

De modo a verificar se a frequência do cristal de oscilação da placa *Arduino* é suficiente para a deteção de cada impulso dos sensores de *hall*, foi necessário determinar a frequência de ocorrência destes para o caso mais crítico, ou seja, à velocidade máxima do motor.

A velocidade máxima de cada motor é de 10000 RPM (rotações por minuto), sendo acoplado ao motor uma caixa redutora de 12:1, que origina uma redução da velocidade máxima aplicada às

rodas, 833 RPM (figura 3.5b). Em cada rotação do motor, ocorrem aproximadamente 293 impulsos elétricos, logo à velocidade máxima de 10000 RPM, que corresponde a 166 RPs (rotações por segundo), ocorrem 166*293=48833 impulsos por segundo, implicando que no máximo os impulsos ocorrem a uma frequência de 48833Hz. Atendendo que o micro-controlador tem de ser capaz de detetar as alterações nas seis portas de interrupção externa, obriga o funcionamento deste a uma frequência superior a 48833*6=293KHz. Uma vez que a frequência de relógio do *Arduino* é de 16MHz, é possível garantir que o sistema irá detetar com segurança cada impulso elétrico.

3.3.2 Configuração das *drivers*

Os motores têm características próprias, independentes de serem do mesmo modelo. Assim foi necessário realizar a configuração das *drivers*, em controlo de malha fechada, para obter os resultados esperados na resposta dos comandos de velocidade.

Portanto procedeu-se à aplicação da interface, disponibilizada pelo fabricante, entre as drivers e o utilizador - *DriverWare7*. Uma das limitações deste software prende-se com o não fornecimento do suporte para a plataforma *linux*, obrigando o utilizador a trabalhar no sistema operativo *Windows* para configurar as *drivers*. Esta configuração passa por realizar três tarefas sequenciais, primeiro a configuração da corrente aplicar nos motores, posteriormente, a auto-comutação do software - ao detetar o tipo de comutação do motor - e por fim o controlo em malha fechada, de modo que o funcionamento das *drivers* seja realizado por comandos de velocidade.

A escolha das drivers para o controlo dos motores *brushless* foi devido aos bons resultados obtidos anteriormente noutras experiências e incluídos no demonstrador de condução autónoma [7]. Outra vantagem destes dispositivos é possuírem protocolos de comunicação RS484/232 e CANopen, já que a comunicação é realizada via porta USB/RS232, garantindo assim uma comunicação segura e suficientemente rápida entre as *drivers* e o ROS.

Configuração da corrente

Na configuração da corrente pretende-se obter o seu comportamento ideal, principalmente no arranque do motor, para os valores definidos no projeto. Este comportamento controlado evita a emissão de ruído, no caso de excessiva corrente e confina o valor da corrente inicial. Esta configuração é realizada pelo controlo de duas constantes: Kp, o ganho proporcional e Ki, o ganho integral, sendo diferente para cada motor. A determinação destas variáveis de controlo é calculada utilizando os valores caraterísticos do motor, como a resistência entre fases, a indutância e a voltagem para afinação da corrente. Define-se o valor limite de um *ampère* em cada motor, como demonstra a figura 3.8.

Esta escolha é motivada pela limitação das baterias que fornecem quatro *ampères*.hora, pela tensão suficiente para exercer um bom arranque e sem excessos de corrente para não alterar a trajetória do robô. Os resultados otimizados para a melhor aproximação ao degrau de um *ampère* são apresentados na tabela 3.3.

Figura 3.8: Configuração malha fechada da corrente

Tabela 3.3: Valores aplicados às *drivers* na configuração da corrente

Configuração da corrente	Motor Esquerdo	Motor Direito
Ganho Proporcional (Kp)	1.504	1.705
Ganho Integral (Ki)	0.318	0.33

Auto-comutação

O processo da auto-comutação, é um dos passos vitais para o funcionamento correto da driver e do motor, já que o software efetua a comparação dos valores introduzidos das características dos motores com os que se encontram montados e ligados à driver. A comutação é dependente do tipo do motor e do *feedback* disponibilizado por este. No caso de motores DC *Brushless*, estes requerem uma comutação trapezoidal para uma correta configuração da *driver*. Esta rotina deteta os sensores de *feedback* acoplados ao motor e verifica-os de acordo com os valores de configuração do motor. Este método identifica o ângulo ótimo entre o campo magnético permanente e o campo eletromagnético, criado pela corrente do motor, deteta os sensores de Hall ou encoders, define a resolução dos sensores e o tipo de comutação obtido, através de duas rotações por um ciclo de impulso elétrico em cada direção. Para o caso dos motores *brushless* utilizados no projeto, o funcionamento correto da comutação corresponde às seguintes características resultantes deste método: deteção de sensores de *hall*, espaçamento dos sensores de *hall* de 120 graus e tipo de comutação trapezoidal. O processo de auto-comutação só fica finalizado com a rotação dos motores em ambos os sentidos [17].

Configuração da velocidade

A configuração da velocidade, em malha fechada, tem como objetivo atingir uma determinada velocidade no menor período de tempo, sem que ocorra dois fenómenos: pico de velocidade e incapacidade de atingir o valor da velocidade pretendida. O pico de velocidade, ocorre quando o motor obtém um excesso de rotações por minuto (RPM) ultrapassando o valor pretendido pela falta de controlo e do *feedback* da informação proveniente dos sensores, podendo danificar componentes e originar resultados inesperados na deslocação do robô. Por outro lado pode não atingir a velocidade pretendida ao longo do tempo e nesta situação é frequente ocorrerem erros consecutivos no deslocamento da trajetória do demonstrador.

Para a configuração da velocidade atingida nos motores em malha fechada, optou-se por realizar como teste a resposta ao degrau de 600 RPM, figura 3.9. Este valor é relativamente baixo, face à capacidade dos motores *brushless*, mas suficiente para o que é pretendido, ou seja, um demonstrador que circule em espaços fechados e relativamente pequenos para efeitos de demonstração. O controlo em malha fechada é realizado internamente na *driver* por um controlo proporcional, integral e derivativo (PID), que recebe a resposta dos motores pelos comandos de velocidade e através dos ganhos do controlador minimiza o erro da velocidade pretendida.

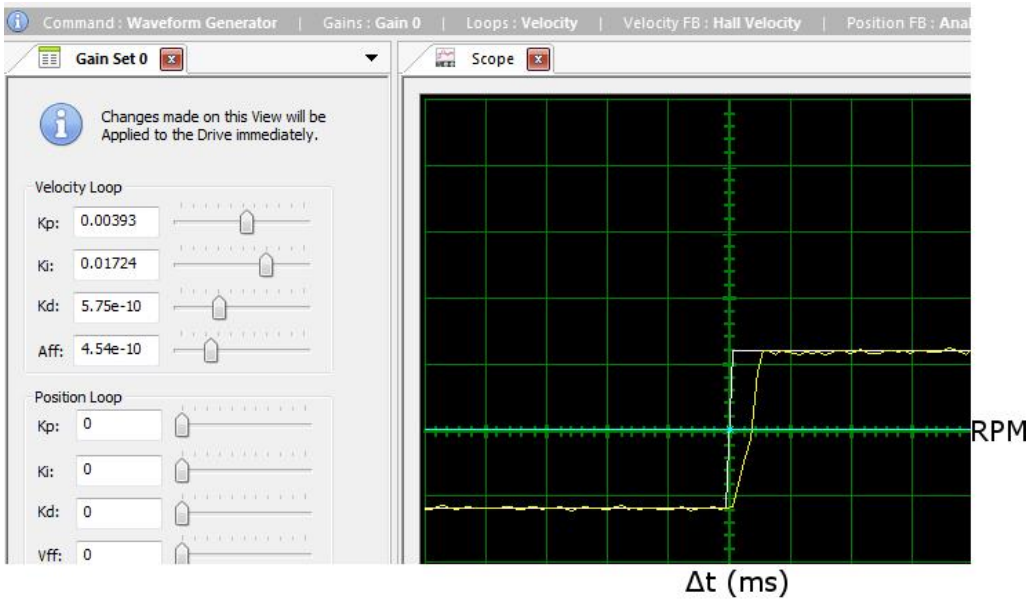


Figura 3.9: Configuração malha fechada da velocidade para o motor esquerdo

Os ganhos do controlador PID, obtidos durante o teste são apresentados na seguinte tabela 3.4.

Tabela 3.4: Valores aplicados às *drivers* na configuração da velocidade

Velocidade malha fechada	Motor Esquerdo	Motor Direito
Ganho Proporcional (Kp)	0.00393	0.00393
Ganho Integral (Ki)	0.01724	0.04642
Ganho Derivativo (Kd)	5.75e-10	4.94e-10

De modo a melhorar a suavização da resposta do motor foi necessário ajustar o filtro da frequência de corte do *feedback* da velocidade. Este filtro, que opera internamente na *driver*, é usado para minimizar as oscilações e o ruído nas medições de velocidade. A frequência de corte do filtro ideal para os motores *brushless* foi de 50Hz.

3.3.3 Protocolo comunicação das *drivers*

A comunicação entre o computador (camada alto nível) e a *driver* (camada *interface*) é utilizada para efetuar configurações, testes, calibrações e controlo dos motores.

O protocolo da comunicação é binário, baseado em *bytes*, do tipo mestre-escravo, em que o papel de mestre é desenvolvido pela aplicação criada pelo fabricante, apresentada na secção 3.3.2 para a configuração inicial, como pelo *package* criado em ROS (secção 3.3.4) para interação dinâmica com o escravo. A comunicação é realizada com comandos que providenciam leitura ou escrita nos parâmetros da *driver*. Cada comando possuí um índice numérico e outras caraterísticas. Faz parte do protocolo a existência de diversos comandos que acedem a um ou mais parâmetros dentro de cada índice numérico. A tabela 3.5 mostra as características da camada física do protocolo.

Tabela 3.5: Características da camada física do protocolo de comunicação

Funções	Características
Comunicação RS232	ponto-a-ponto, nó simples
Máximo *Baud rate*	921600 bits/s
Baud rate pré-definido	9600 bits/s
Intervalo dos endereços dos nós	1 a 63
Endereço do nó pré-definido	63

Na figura 3.10 estão ilustrados os vários constituintes do comando de leitura/escrita. A informação da função do comando é definido no parâmetro *Index* e o tipo de mensagem no *control byte*.

Nos tópicos seguintes é explicado o significado de cada parcela da mensagem enviada pelo mestre.

***Start of frame* (SOF)**

Todas as mensagens entre o mestre e escravo começam pelo *byte* SOF. O *byte* SOF em hexadecimal é sempre A5h, independentemente, da mensagem ser enviada pelo mestre ou escravo.

Address

Address é uma parcela da mensagem destinada ao endereço do nó. O endereço, por defeito de fabrico, é o 63 que corresponde em hexadecimal a 3Fh. A tabela 3.6 mostra o leque de endereços e a sua validade.

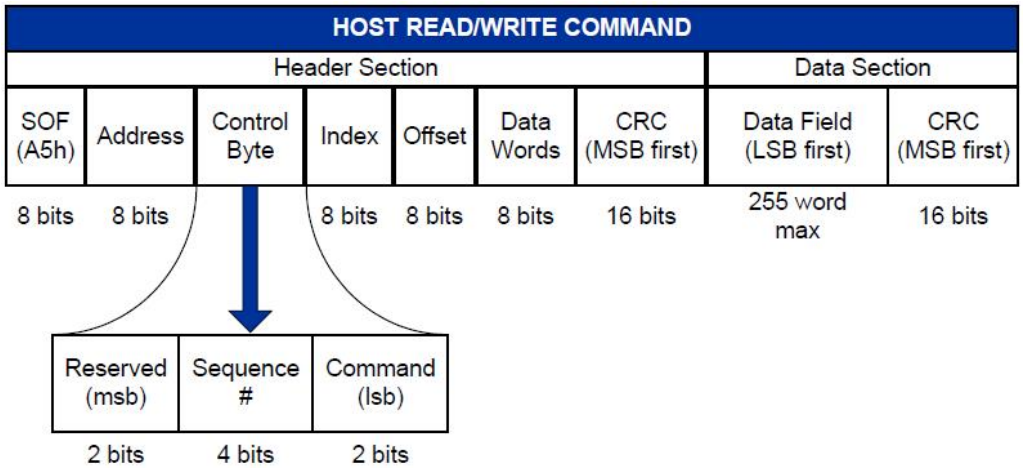


Figura 3.10: Configuração da mensagem a ser enviada pelo mestre [9]

Tabela 3.6: Lista de endereços dos nós [12]

Número do endereço	Descrição
00h	Mensagem de comando enviado do *host* para todas as *drivers*.
01h-3Fh	Endereço do nó valido. *Host* apenas comunica com uma *drive* de cada vez.
40h-FEh	Endereço ilegal.
FFh	Reservado para endereço mestre. Todos os nós respondem com o endereço FFh.

Control Byte

O *Control Byte* é usado para especificar o tipo de mensagem e sequência de controlo. A tabela 3.7 mostra em detalhe a descrição para o *control byte*.

Os *bits* de sequência no comando, enviado pelo mestre, podem assumir qualquer tipo de número e a trama de resposta, terá de ter obrigatoriamente, o mesmo número de sequência. Estes *bits* podem ser sequenciais ou fixos e caso o número da sequência não corresponda, a mensagem de resposta será ignorada.

Index

As operações básicas da *driver* AMC encontram-se numa lista de índices, em que cada um destes números contém parâmetros. Este formato tipo vetor é apresentado na figura 3.11.

No anexo A é apresentada em detalhe a lista de *Index* para aceder aos dados guardados na memória da *driver* AMC.

Tabela 3.7: Descrição do byte de controlo [12]

Comando Bits 0 e 1	Sequência Bits 2 - 5	Reservado Bits 6 e 7	Descrição
0	Critério do utilizador	00	Reservado para utilização futura.
1	Critério do utilizador	00	Esta mensagem não contém dados. A mensagem de resposta irá conter o número de *words* específicos no *byte data words* para uma localização especificada pelo *byte offset*.
2	Critério do utilizador	00	Esta mensagem contém o número de *words* especifico na sequência no comando *data words* para uma localização especificada pelo *byte offset*. A mensagem de resposta não irá conter dados.
3	Critério do utilizador	00	Reservado para utilização futura.

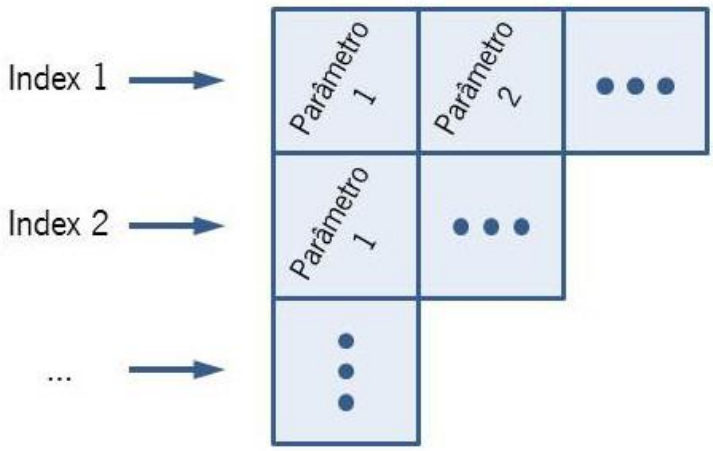


Figura 3.11: Formato tipo vetor dos índices e parâmetros

Offset

O *offset byte* é usado para identificar o parâmetro de um *index* especifico. O valor do *byte* indica qual é a distância do primeiro parâmetro do índice ao parâmetro que se pretende aceder.

Data Words

É o valor que indica o número de *words* (2 *bytes*) na parcela *Data Field*. Os dados não podem ultrapassar as 255 *words* (510 *bytes*), consequentemente o intervalo de valores permitidos são de 0 a 255.

No caso de um comando de escrita, *Data Words* indica o número de *words* a receber na mensagem pelo escravo. No caso de um comando de leitura, *Data Words* indica o número de *words* na mensagem de resposta do nó.

Data Field

Contém os dados a escrever ou a ler na memória da *driver* AMC. O tamanho do campo *Data Field* é do seguinte formato:

- Contém um determinado número de *bytes* no caso de comando de escrita;
- Não contém dados no caso de comando de leitura;
- Dados são sempre apresentados e guardados no formato Little Endian - LSB first (Least Significant Bit);
- Campo de dados máximo permitido 510 *bytes* (255 *words*).

Header CRC e *Data* CRC

Ambos *Header Section* e *Data Section* devem conter um valor no Cyclic Redundancy Check (CRC). CRC é método para validar os dados enviados nos comandos e permite detetar anomalias nas mensagens transmitidas. Se não existirem dados no campo *Data Field*, os parâmetros do CRC da *Data Section* são descartados.

Se o escravo não identificar o endereço ou o valor do CRC do *Header Section* não corresponder, a mensagem será ignorada até detetar um novo SOF. Caso passe as validações anteriores e falhe na verificação do CRC do Data Section, será enviada pelo escravo, uma mensagem de erro.

O método de verificação do CRC é referido como CRC-16-CCITT e é baseado no polinómio $X^{16}+X^{12}+X^{5}+1$. A criação e apresentação da tabela com os valores do CRC é apresentada no anexo B.

Mensagem de resposta

A resposta do escravo encontra-se ilustrada na figura 3.12.

As principais alterações para a mensagem do mestre são em dois campos do *Header Section* - *Address* e *Status 1*. O escravo responde sempre FFh no *byte Address*.

A diferença no *Status 1* é o facto da mensagem de resposta informar o mestre da ocorrência de alguma anomalia (tabela 3.8).

O campo *Status 2* não é definido na mensagem de resposta.

3.3.4 Sistema ROS Locomoção

O sistema de locomoção em pleno funcionamento no ROS, começa com a receção dos impulsos elétricos no *Arduino Mega*, onde publica o total de impulsos de ambos os motores no tópico do sistema ROS. Com base nos impulsos elétricos do sensor *hall*, realiza-se o calculo da posição,

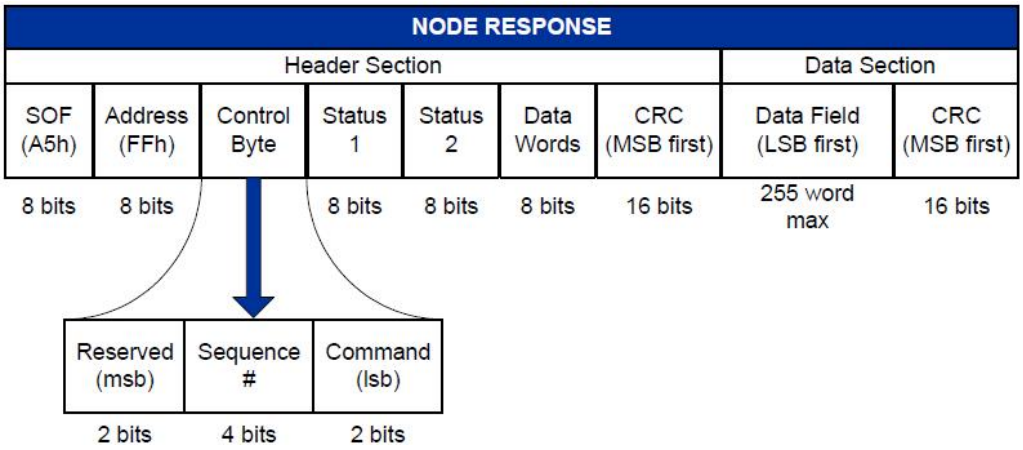


Figura 3.12: Configuração da mensagem a ser enviada pelo escravo [9]

Tabela 3.8: Tabela de interpretação do parâmetro *Status 1* [12]

Valor	Descrição
1h	Comando completo.
2h	Comando incompleto.
4h	Comando inválido.
6h	Sem permissão de acesso de escrita.
8h	Trama ou CRC erro.

velocidade ocorrida e direção do demonstrador. Estas operações são realizadas no pacote *Controlo* e são responsáveis pela decisão lógica da trajetória do robô, que iremos abordar na secção 3.4.

Para criar o ciclo do sistema de locomoção, desde a deteção dos impulsos elétricos até ao envio de comandos de velocidade aos motores, foi desenvolvido um pacote, denominado por *DriverDZR*, que controla a capacidade de locomoção do robô - *drivers* e motores. O ciclo do funcionamento do sistema de locomoção apresenta-se na figura 3.13.

Funcionamento *Arduino* no ROS

Para o funcionamento do *Arduino* no ROS, recorreu-se ao pacote *rosserial*, disponibilizado na comunidade ROS. Este pacote interage e realiza a comunicação entre *Arduino* e ROS, dando assim acesso à publicação e subscrição nos tópicos do ROS.

Na placa *Arduino* ocorre a publicação dos impulsos elétricos na plataforma ROS. Para este tipo de mensagem optamos por um vetor de inteiros sem sinal de oito *bits* (*std_msgs::UInt8MultiArray*), para enviar o conjunto dos impulsos dos dois motores para o tópico */odo_C2*. O motivo que levou à escolha deste tipo de dados foi o reduzido espaço, que ocupa na memória do *Arduino* e que envolve números inteiros positivos entre 0 a 65535. Como este valor é facilmente ultrapassado em poucos metros de movimentação do demonstrador foi necessário assegurar, com recurso à programação, que o valor excedente, seja automaticamente corrigido e capaz de continuar a fornecer a contagem dos impulsos corretos da hodometria.

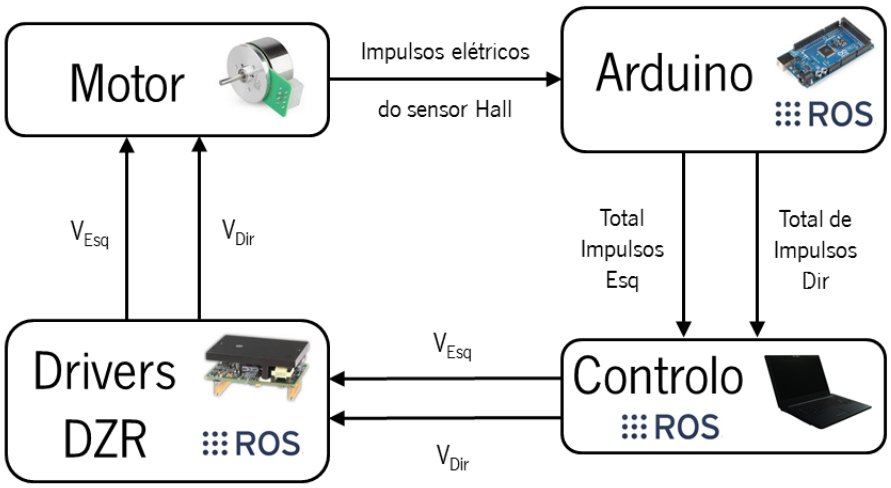


Figura 3.13: Funcionamento do sistema de locomoção

Na figura 3.14 apresenta-se, através da ferramenta *ROSgraph*, os tópicos subscritos e a ser publicados no nó */serial_node*.

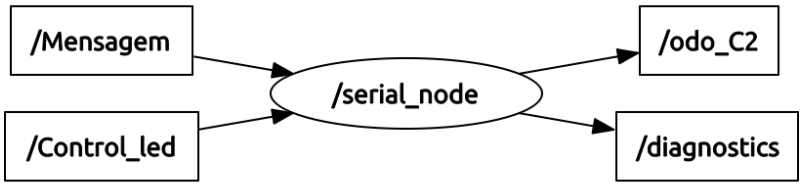


Figura 3.14: *ROSgraph* do *Arduino*

O nó recebe dados das subscrições aos tópicos */Mensagem* e */Control_led*, que se destinam ao controlo dos LEDs abordados na subsecção 3.5 e publica para o tópico */odo_C2* a contagem de impulsos elétricos dos dois sensores *hall*. O tópico */diagnostics* do pacote *rosserial*, destina-se para controlo da sincronização entre o *Arduino* e o servidor ROS.

Funcionamento das *Drivers* no ROS

Na realização do porte do sistema de locomoção para ROS, foi também desenvolvido o pacote *DriverDZR*, responsável por toda a comunicação via rs-232, entre computador e as *drivers*. Na elaboração deste pacote, foi criado um conjunto de requisitos de forma a clarificar as metas atingir:

- Verificar a iniciação do servidor ROS (roscore);
- Confirmar a identificação da driver e número de dispositivo;
- Alterar os parâmetros em tempo de execução;

- Obter acesso de escrita e envio de comandos de velocidade por rs-232;
- Obter uma comunicação robusta e eficiente com a *driver*;
- Informar de qualquer anomalia detetada ao utilizador no decorrer do programa;
- Iniciar e configurar os parâmetros pré-definidos das *drivers*.

O pacote foi programado para detetar o servidor ROS no início da aplicação e caso este não seja detetado, avisa o utilizador e fica a aguardar pela inicialização do servidor do ROS (*roscore*). Após executar o pacote no servidor do ROS, são iniciadas as comunicações e configurações das *drivers*.

A iniciação da *driver* passa pela abertura da porta *Universal Serial Bus* (USB) recorrendo à biblioteca *QextSerialPort*. O *QextSerialPort* é uma biblioteca com licença MIT - *open source initiative*, em que providencia a ligação com porta série e porta USB, para aplicações baseadas no IDE (*Integrated development environment*) do *QtCreator*. Após o início da comunicação entre a porta USB (computador) / RS232 (*driver*) dá-se início ao procedimento da identificação das *drivers*. Esta identificação processa-se através da comparação entre os nomes da configuração e dos endereços armazenados na memória da *driver* com as configurações guardadas no disco rígido. Realizada esta operação, é enviado o comando que permite o acesso à escrita nas drivers (*SetAccessControl*), e na inoperacionalidade desta situação, só é possível o envio de comandos de leitura impedindo deste modo a ativação da *bridge*.

Após a ligação e o acesso à escrita na driver, procede-se à configuração das seguintes tarefas:

- Ativar a *Bridge*;
- Ativar a paragem dos motores, ou seja, impedir a iniciação do arranque dos motores;
- Definir o tempo máximo de resposta (*Heartbeat*) entre mestre e escravo, que neste projeto é de 500 milissegundos. Caso o *Heartbeat* seja excedido e sem existir qualquer tipo de comunicação entre os dois nós, a *bridge* é desligada.

Com o termino da configuração, completa-se a iniciação das *drivers*, como demonstra a figura 3.15. Esta representa a inicialização na consola do computador com a ligação da porta USB, apenas conectada pós executar o pacote, fazendo a deteção automática das *drivers*.

Como foi referido anteriormente, a configuração do arranque da *driver* é guardada no disco rígido, utilizando o pacote *dynamic_reconfigure*, disponibilizado na plataforma ROS. Este pacote permite guardar parâmetros no disco rígido e alterar os mesmos, sem que seja necessário reiniciar o nó. Assim, a qualquer momento e a título exemplificativo, pode-se alterar a porta USB da *driver* ou o nome da configuração da mesma.

Os parâmetros de configuração, que são utilizados neste projeto são guardados em ficheiros *cfg* e apresentam-se na tabela 3.9.

A figura 3.16 ilustra a janela gráfica, que permite alterar parâmetros da *driver* sem ter que desligar ou reiniciar a comunicação com a mesma.

Após a ligação ao servidor ROS, o pacote *driverDZR* subscreve-se ao tópico */cmd_vel*, onde recebe valores de velocidade providenciados pelo sistema de controlo. *Twist*, foi o tipo de mensagem selecionado neste tópico, pois é um formato de mensagem do ROS, idealmente, utilizado

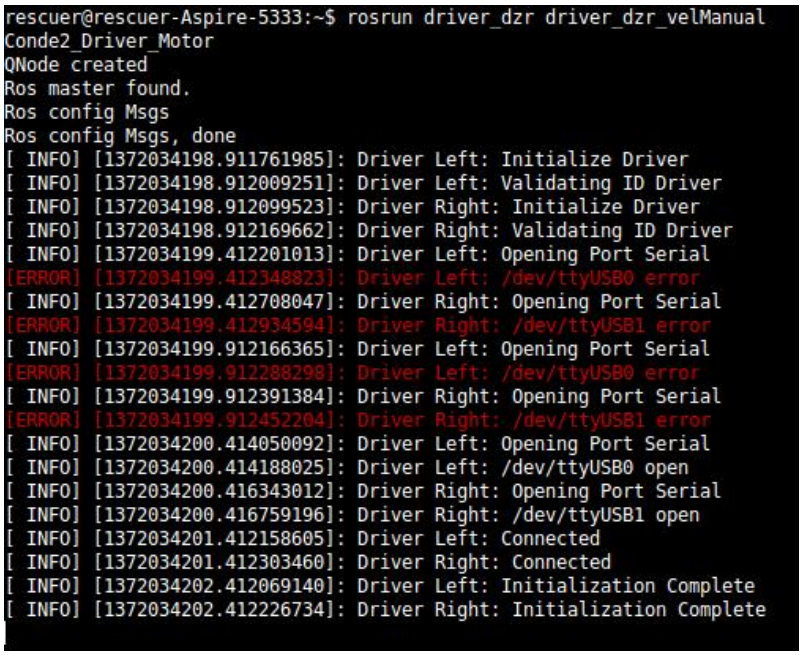


Figura 3.15: Iniciação do pacote *driverDZR*

Tabela 3.9: Parâmetros guardados no disco rígido

Variável	Formato	Descrição
SerialPortNameLeft	*string*	Define a porta USB da *driver* esquerda
DriverNameLeft	*string*	Define o nome da *driver* esquerda
SerialPortNameRight	*string*	Define a porta USB da *driver* direita
DriverNameRight	*string*	Define o nome, que por defeito é *"C2_right"*, para a *driver* direita
WheelRadius	*double*	Distância do raio das rodas
DistanceOfWheels	*double*	Distância entre rodas

para comandos de velocidade (*geometry_msgs::twist*). O *Twist* é constituído por dois vetores, um para a velocidade linear e outro para a velocidade angular, como é descrito na tabela 3.10.

Tabela 3.10: Constituição da mensagem *Twist*

Vetor	Eixos	Descrição
linear	x	utilizado para indicar a velocidade linear (v)
	y	componente linear
	z	componente linear
angular	x	componente angular
	y	componente angular
	z	utilizado para indicar a velocidade angular (w)

Na figura 3.17, ilustra-se na ferramenta *ROSgraph*, o funcionamento do nó *Conde2_Driver_Motor* do pacote *driverDZR*.

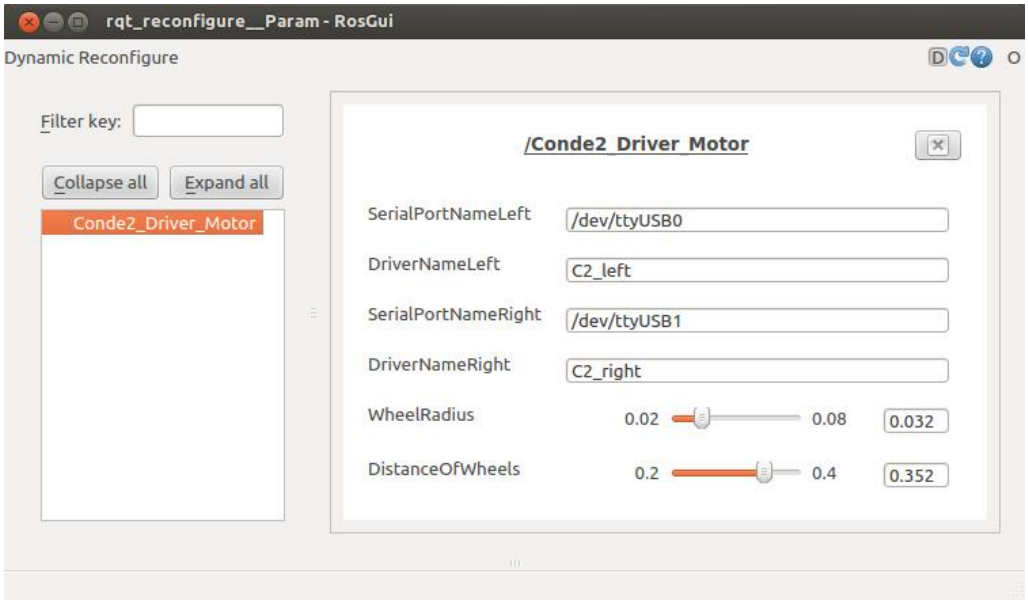


Figura 3.16: Janela alteração configuração da *driver* - *dynamic_reconfigure*

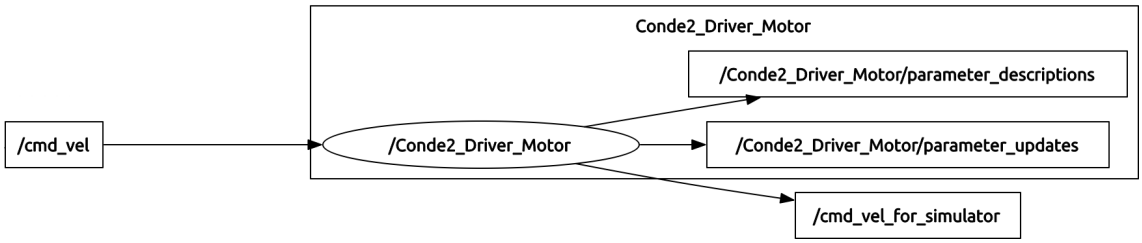


Figura 3.17: *ROSgraph* da *driver*

O tópico */cmd_vel_for_simulator*, como o nome indica, realiza a propagação da velocidade real dos motores para o simulador *stage*. Este tópico também no formato *twist*, publica a informação da velocidade do robô, baseando-se no *feedback* dos impulsos elétricos fornecidos pela plataforma *Arduino*. Esta operação do cálculo da velocidade através dos impulsos, ao invés de publicar a velocidade proveniente do sistema de controlo, evita que erros não sistemáticos, como o deslizamento das rodas, não afete o comportamento do demonstrador no simulador.

3.4 Sistema de Controlo

Nesta secção aborda-se a metodologia utilizada para obtenção da posição do robô em malha fechada, o cálculo da velocidade e direção do robô. Foca-se também o controlo por seguimento de trajetórias.

O funcionamento do sistema de controlo apresentado na figura 3.18, mostra como se realiza todo o processo, desde a obtenção dos valores da hodometria até ao envio dos valores de velocidade para as *drivers* e que será descrito nas subsecções seguintes.

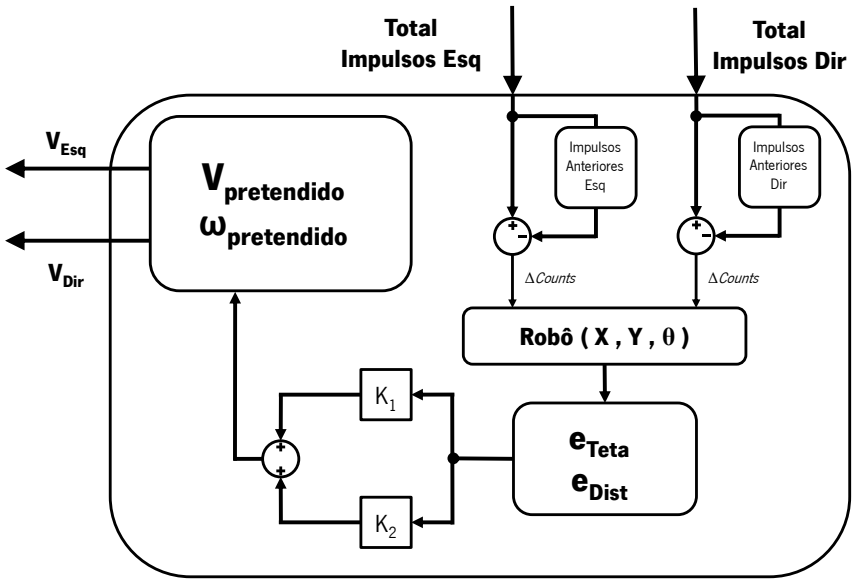


Figura 3.18: Funcionamento do sistema de controlo

3.4.1 Modelo da Hodometria

A hodometria é um dos métodos, amplamente, utilizados para estimar a posição de um robô. Este método de localização relativa, proporciona uma boa precisão em curto prazo, é de baixo custo a sua implementação e permite altas taxas de amostragem. O propósito da hodometria é a integração da informação sobre movimentos incrementados ao longo do tempo, o qual envolve uma inevitável acumulação de erros. Esta acumulação causa grandes desvios na estimação da posição do robô, os quais vão aumentando, proporcionalmente, com a distância percorrida.

Os erros de hodometria podem sub-agrupar-se em dois tipos: erros sistemáticos e não sistemáticos. Os primeiros são relacionados com as caraterísticas do robô e/ou dos sensores, como por exemplo as imprecisões no desalinhamento e no diâmetro das rodas. Os erros não sistemáticos são relacionados com a *interface* entre o robô e o ambiente, nomeadamente os tipos de solo não uniformes (rugosos, escorregadios, etc), os obstáculos inesperados e o escorregamento das rodas [18][19].

O modelo da hodometria permite, a partir da informação dada periodicamente pelos sensores *hall* de cada roda, medir a posição do robô entre intervalos de amostragem.

A fim de determinar a posição do robô sobre um plano, temos que estabelecer uma relação entre o eixo de coordenadas globais do plano e o eixo de coordenadas locais do robô, tal como exibe a figura 3.19. A especificação do ponto da posição, P, corresponde ao ponto médio do segmento de reta da distância entre eixos [20].

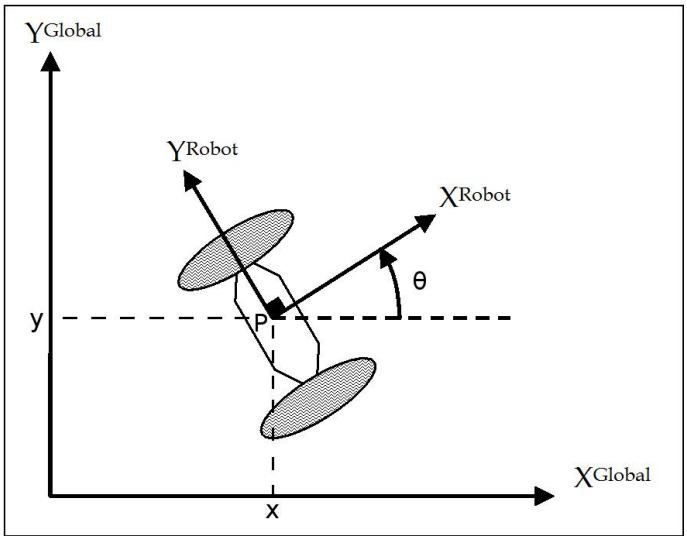


Figura 3.19: Eixos de coordenada global e local [10]

A base (X_{robot},Y_{robot}) define dois eixos em relação ao ponto P e é portanto o eixo de coordenadas locais. O ponto P no eixo de coordenadas globais é identificado pelas coordenadas x e y, e a diferença de ângulo entre o eixo global e local é dado por θ. Podemos descrever a posição e orientação do robô como um vetor de três elementos (expressão (3.1)).

$$P = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{3.1}$$

A diferença de impulsos entre os intervalos de tempo da medição, obtém-se pela subtração dos impulsos elétricos provenientes no preciso momento da rotação de cada motor, com os valores guardados da medição anterior. Esta diferença de impulsos elétricos ($\Delta Counts$), juntamente com o valor de impulsos por metro (ΔIpM), equação (3.3), permite obter a distância percorrida por cada roda do robô em cada período de tempo, como é observável na equação (3.2).

$$\Delta Dist_n = \frac{\Delta Counts_n}{\Delta IpM_n} \tag{3.2}$$

$$\Delta IpM_n = \frac{\Delta IpR_n \times n}{2\pi \times R_{e/d}} \tag{3.3}$$

Em que $R_{e/d}$, ΔIpR_n e n são respetivamente o raio das rodas, a resolução do sensor (impulsos por revolução) e a relação da caixa redutora.

Com o conhecimento do período de tempo entre a qual foi medida a distância percorrida por cada roda, equação (3.4), obtém-se a velocidade de cada motor.

$$v_n = \frac{\Delta Counts_n}{\Delta IpM_n \times \Delta t} \tag{3.4}$$

Das equações de cinemática (3.5) (3.6) retira-se a velocidade linear (v) e velocidade angular do robô (ω). A distância entre eixos ($dist_{axis}$), num sistema diferencial, corresponde à distância entre as rodas que influência inversamente a velocidade angular.

$$v = \frac{v_{dir} + v_{esq}}{2} \tag{3.5}$$

$$\omega = \frac{v_{dir} - v_{esq}}{dist_{axis}} \tag{3.6}$$

Conhecidas as velocidades obtidas pelo *feedback* dos sensores de *hall* ($\Delta Counts$), calcula-se a posição global do robô em relação ao eixo de coordenadas do mapa. A posição do robô no mapa, consiste no ponto de coordenadas (X_n, Y_n) - equações (3.7) (3.8) - e o ângulo de orientação do robô (θ_n) - equação (3.9).

$$X_n = X_{n-1} + v \times cos(\theta_{n-1} + \frac{\omega}{2})\Delta t \tag{3.7}$$

$$Y_n = Y_{n-1} + v \times sin(\theta_{n-1} + \frac{\omega}{2})\Delta t \tag{3.8}$$

$$\theta_n = \theta_{n-1} + \omega\Delta t \tag{3.9}$$

3.4.2 Controlo com base na posição

No estudo desta dissertação foram desenvolvidos controlos por seguimento de reta, de ponto e de ângulo. Estes controlos são com base na posição do robô (x,y).

Uma trajetória, mesmo que complexa, pode ser entendida como a segmentação de inúmeros segmentos de retas. A utilização desta abordagem visa reduzir a carga computacional necessária para a movimentação do demonstrador, visto que para determinar uma reta é suficiente a definição de dois pontos no espaço. Além disso, o uso de segmentos de reta possibilita a definição e ajuste das velocidades linear e angular em cada ponto do segmento, tendo em consideração os valores correspondentes ao ponto inicial e final.

Os controladores apresentados nesta secção possuem uma realimentação parcial do estado do robô (x,y,θ), uma vez que corrige a posição e a orientação no plano cartesiano. No estudo desta dissertação foram implementados três controladores que utilizam esta técnica: controlo por seguimento de pontos, retas e ângulo [21].

3.4.2.1 Seguimento de reta

O controlo por seguimento de reta não é utilizado para conduzir o robô a determinado ponto, mas para manter-lo guiado sobre um segmento de reta. Portanto, considera-se a trajetória pre-

tendida S, que contém o segmento de reta definidos pelos pontos A (x_a,y_a) e B(x_b,y_b). A figura 3.20 demonstra as representações da posição, orientação do demonstrador e da trajetória S do seguimento.

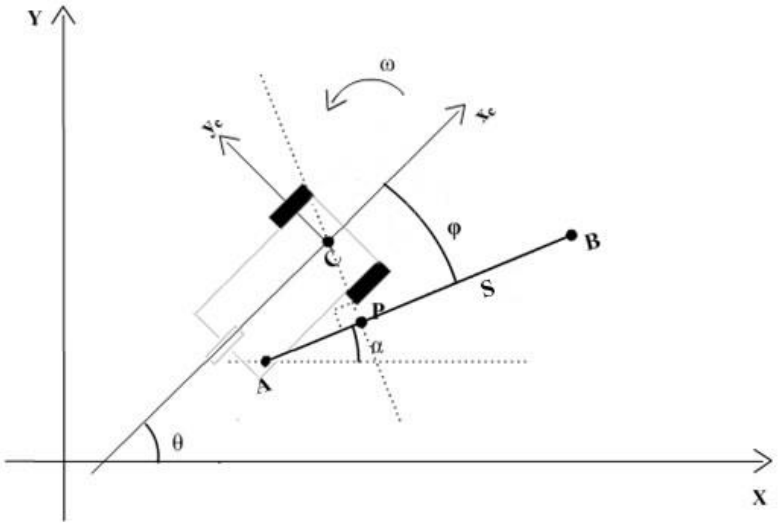


Figura 3.20: Representação esquemática do seguimento de reta

O ângulo φ é denominado de erro de orientação do demonstrador, expressão 3.10, e consiste na diferença entre o ângulo α do segmento $\overline{AB}$ e o ângulo θ do robô no sistema de coordenadas globais.

$$\varphi = \theta - \alpha \tag{3.10}$$

Onde o cálculo do ângulo α, do segmento de reta $\overline{AB}$, é dado pela equação 3.11.

$$\alpha = arctan(y_b - y_a, x_b - x_a) \tag{3.11}$$

A velocidade angular de referência (ω), leva em consideração, além da componente de erro de orientação, o erro e_{linha} que representa a distância mínima entre o ponto C e o segmento de reta $\overline{AB}$. Sendo S definido pela equação da reta: $ax_r + by_r + c = 0$, então e_{linha} pode ser calculado pela equação 3.12.

$$e_{linha} = \frac{ax_c + by_c + c}{\sqrt{a^2 + b^2}} \tag{3.12}$$

Para o controlo da rotação do demonstrador, neste tipo de seguimento, realiza-se a o cálculo presenta na equação 3.13.

$$\omega = -k_1\varphi - k_2 e_{linha} \tag{3.13}$$

Em que o k_1 e k_2 representam os ajustes do ganho proporcional.

A velocidade linear (v) do seguimento de reta é dado pela equação 3.14, onde v_n é a velocidade nominal do robô.

$$v = v_n + k_2 e_{dist} \tag{3.14}$$

O erro e_{dist}, apresentado na expressão 3.15, consiste na distância entre a projeção do ponto C sobre a reta $\overline{AB}$ e o ponto B. Este erro é incumbido por incrementar ou decrementar a velocidade linear, respetivamente, para o afastamento ou a aproximação ao ponto B.

$$e_{dist} = \sqrt{(x_b - x_p)^2 + (y_b - y_p)^2} \tag{3.15}$$

3.4.2.2 Seguimento de ponto

No controlo por seguimento de ponto, o objetivo é que o demonstrador realize a deslocação da sua posição C (x_c,y_c) até a um determinado ponto objetivo B (x_b,y_b). Para isso, é traçado uma trajetória S composta por um segmento de reta $\overline{CB}$ de orientação α definidos no eixo de coordenadas globais. Este método pode ser entendido como uma simplificação do controlo por seguimento de retas, uma vez que o demonstrador se mantém sobre o segmento de reta que é recalculado a cada ciclo com os dados da posição atualizada. A figura 3.21 mostra a representação esquemática deste controlador.

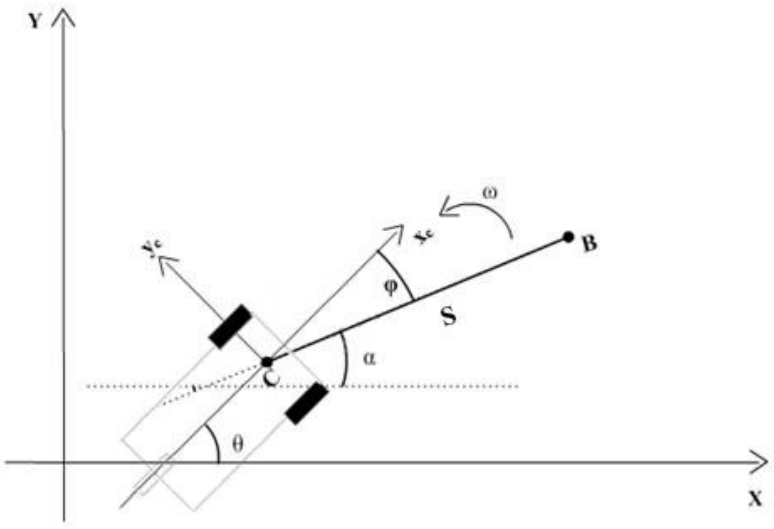


Figura 3.21: Representação esquemática do seguimento de ponto

Para este controlador o erro de orientação φ, mantém a sua relação entre α e θ como no controlo por seguimento de reta. Entretanto, a definição do ângulo α sofre uma pequena alteração pois será formado entre o segmento $\overline{CB}$ e o eixo de coordenadas globais (equações 3.16, 3.17).

$$\varphi = \theta - \alpha \tag{3.16}$$

$$\alpha = arctan(y_b - y_c, x_b - x_c) \tag{3.17}$$

Neste modelo de seguimento o erro ao segmento de reta e_{linha} é nulo, uma vez que o ponto C faz parte da trajetória S. Portanto, a velocidade angular do robô é definida pela equação 3.18, que utiliza apenas o erro de orientação e o ganho proporcional k_1.

$$\omega = -k_1 \varphi \tag{3.18}$$

A abordagem ao cálculo da velocidade linear na expressão 3.19, faz-se considerar pelo erro de distância (e_{dist}) do ponto C ao ponto B, equação 3.20.

$$v = v_n + k_2 e_{dist} \tag{3.19}$$

$$e_{dist} = \sqrt{(x_b - x_c)^2 + (y_b - y_c)^2} \tag{3.20}$$

3.4.2.3 Seguimento de ângulo

O controlador para seguimento de ângulo possui uma lógica bastante simplificada, porém não menos importante que os outros. É utilizado em situações onde é necessário um movimento puramente angular do demonstrador, como na orientação inicial de trajetórias e em certas correções de percursos. Este controlo pode ser deduzido como uma generalização do controlo por seguimento de ponto, onde o robô encontra-se sobre o ponto objetivo mas sem a orientação correta.

O ângulo φ necessário para a correção da orientação do demonstrador é dado pela equação 3.21.

$$\varphi = \theta - \alpha \tag{3.21}$$

Onde θ corresponde ao ângulo da posição do robô no eixo de coordenadas globais e o α o ângulo de posição pretendido, como ilustrado na figura 3.22.

Neste controlador a velocidade linear é nula pois pretende-se uma rotação sem translação e o cálculo da velocidade angular de referência (ω), que é demonstrado pela equação 3.22, onde k_1 é o ganho de ajuste.

$$\omega = -k_1 \varphi \tag{3.22}$$

Fluxograma do sistema de controlo

O diagrama correspondente aos procedimentos do sistema de controlo está ilustrado na figura 3.23.

Como se pode verificar pela figura 3.23, após o arranque do sistema o processo entra num ciclo infinito, onde é feita a atualização da hodometria e o cálculo da diferença de impulsos e permanecendo neste ciclo até que $\Delta Counts$ seja diferente do último valor guardado.

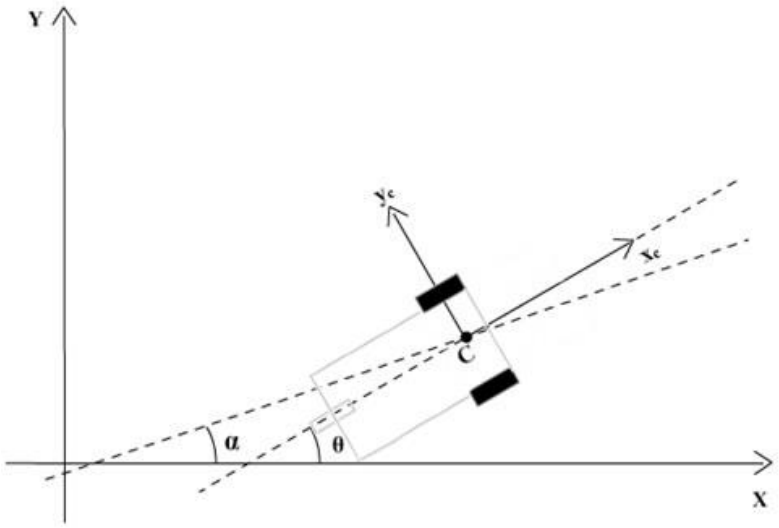


Figura 3.22: Representação esquemática do seguimento de ângulo

Após a alteração da diferença de impulsos ($\Delta Counts$), o sistema atualiza a posição e orientação do robô que será utilizada para calcular se alcançou a posição final do seguimento da trajetória. Caso isso aconteça é ordenado paragem do demonstrador e termina-se o ciclo do sistema de controlo. Consequentemente, se o robô não tiver atingido a posição final o sistema entra noutro ciclo responsável pelo cálculo dos erros do controlo de seguimento e pela velocidade linear e angular pretendidas (v_{pret}, ω_{pret}) para corrigir a trajetória do demonstrador.

O cálculo dos erros da trajetória e controlo das velocidades pretendidas, foram apresentadas nas secções anteriores (3.4.2.1, 3.4.2.2 e 3.4.2.3). O último bloco, posterior à verificação da sincronização com as drivers, é responsável por limitar às velocidades nominais constantes, respetivamente, linear e angular.

O envio do comando de velocidade só é realizado se a comunicação entre o sistema de controlo e o sistema de locomoção estiver sincronizado, ou seja, a receber constantemente numa frequência superior a 3.3 Hz (menor que 300ms). Esta frequência é responsável por impedir o robô de progredir caso perca-se a ligação com a *driver* e sendo inferior a 300 ms, permite obter espaço temporal suficiente para atuar nos motores em futuras aplicações, como mudanças de trajetória para desvio de objetos.

3.4.3 Sistema ROS de controlo

No porte do sistema de controlo para ROS, foi implementado o pacote *C2_control* que realiza o cálculo da posição, orientação do robô e corrige a movimentação do demonstrador para o seguimento da trajetória pretendida. Estas informações produzidas são publicas em vários tópicos, consoante o seu tipo de mensagem, como ilustrado na figura 3.24.

O sistema de controlo recebe a contagem dos impulsos proveniente do tópico */odo_c2* (a correr no *Arduino*) e após o processamento dos dados publica no servidor do ROS o posicionamento do

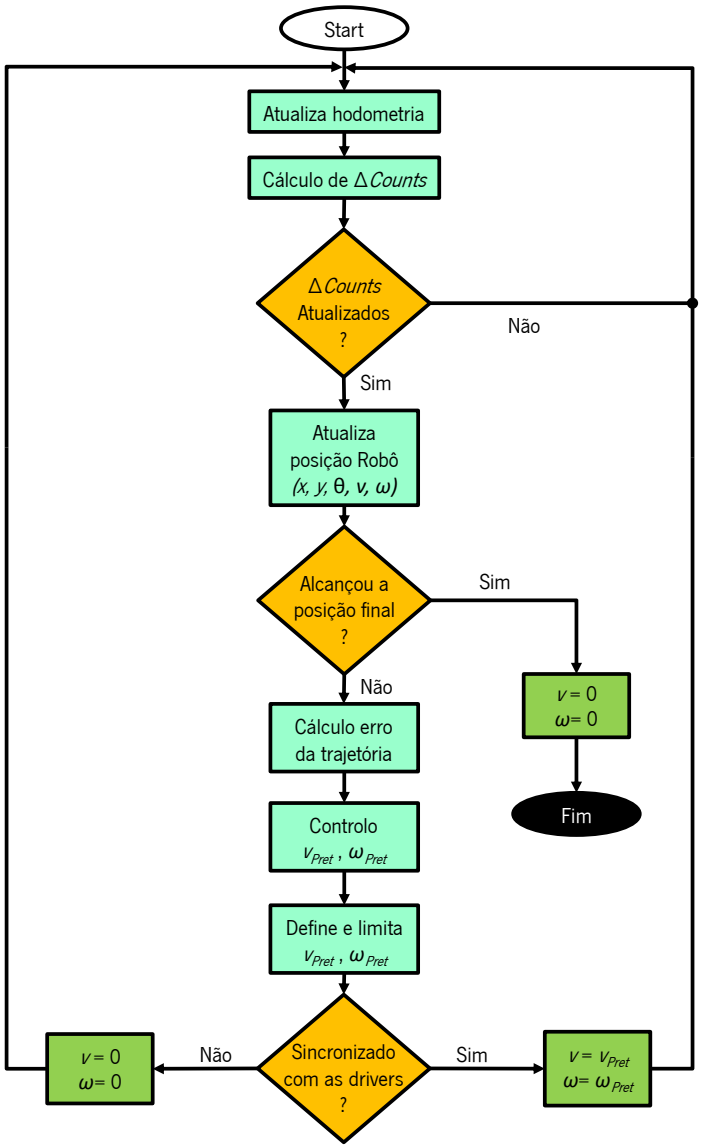


Figura 3.23: Fluxograma do sistema de controlo

robô (posição e orientação), o total e a diferença de impulsos elétricos, os comandos de velocidade e múltiplas coordenadas ao longo do tempo (*transform frames*).

Transform frames, denominado por *tf*, é um tipo de mensagem do ROS que faz a manipulação do relacionamento no tempo entre o sistema de coordenadas com a estrutura árvore (*tree structure*), ou seja, transformações de pontos, vetores, etc entre dois pontos do sistema de coordenadas e qualquer ponto desejado no tempo.

A manipulação deste tipo de mensagem requer a criação de uma estrutura árvore, que relaciona todos os pontos do robô ao mapa do simulador. O tipo de mensagem *tf* é requerido para a comunicação entre vários serviços do ROS, como exemplo o simulador *stage*.

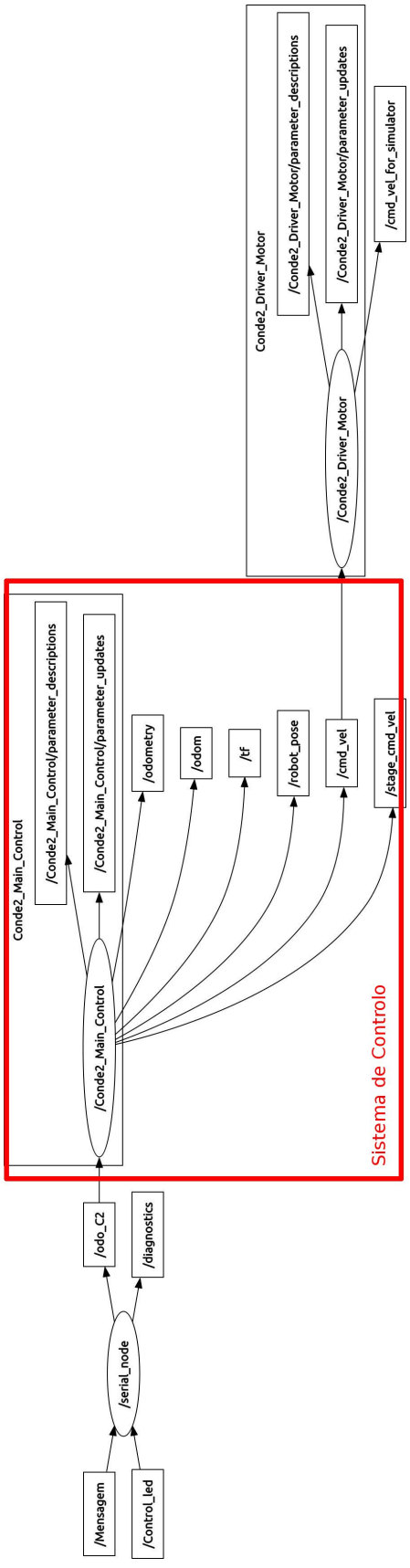


Figura 3.24: *ROSgraph* da interação do sistema de locomoção com o sistema de controlo

A elaboração, realizada neste projeto, da *tree structure* do demonstrador é desenvolvida com recurso ao pacote *URDF* (*Unified Robot Description Format*). O resultado final da criação do ficheiro *.urdf* para este projeto é demonstrado na figura 3.25, como apresentamos no anexo E a sua constituição.

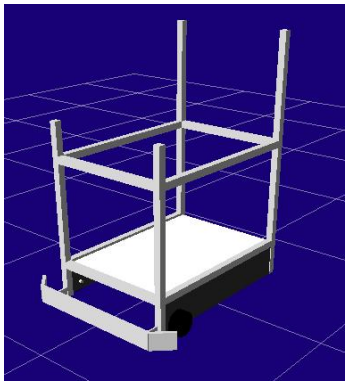


Figura 3.25: Visualização do modelo do robô na ferramenta *rviz*

Na tabela 3.11, faz-se correspondência do tópico do ROS com o tipo de mensagem e a sua função.

Tabela 3.11: Tópicos do sistema de controlo no ROS

Tópico ROS	Tipo de mensagem	Descrição
/odo_C2 (subscreve)	*std_msgs::UInt32MultiArray*	Recebe os impulsos elétricos
/odometry (publica)	*std_msgs::String*	Envio da hodometria num formato *string*
/cmd_vel (publica)	*geometry_msgs::Twist*	Envio de comandos de velocidade
/stage_cmd_vel (publica)	*geometry_msgs::Twist*	Publica comandos de velocidade para simulador
/tf (publica)	*tf::TransformBroadcaster*	Envio da mensagem tipo *tf*
/odom (publica)	*nav_msgs::Odometry*	Publica no formato de navegação do ROS o valor da hodometria
/robot_pose (publica)	*geometry_msgs::Pose2D*	Envio da posição e orientação do robô

3.5 Sistema de LEDs

O sistema de LEDs RGB (*Red, Green e Blue*) foi desenvolvido para realizar operações de iluminação, como também pode funcionar como uma ferramenta de testes para identificar procedimentos ocorridos no demonstrador. A placa desenvolvida para o sistema de iluminação (figura 3.26), contém oito LEDs RGB, um regulador de tensão 24/5V DC, três circuitos integrados (ULN2803) e vários componentes elétricos.

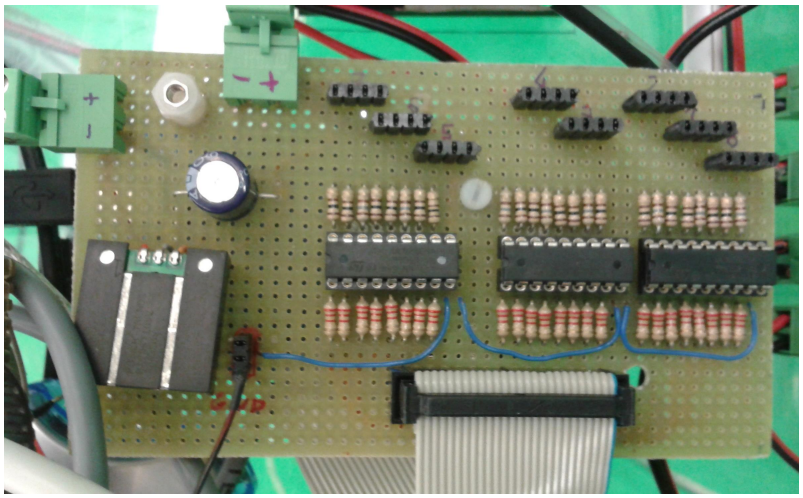
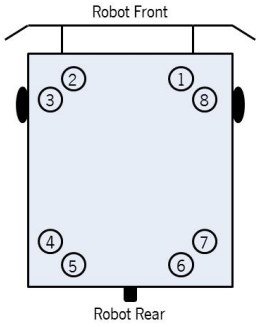
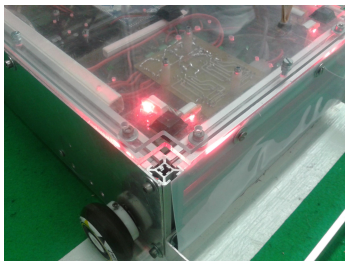


Figura 3.26: Placa do sistema de LEDs

Os LEDs foram colocados trespassados na placa de acrílico e dois em cada canto do *chassis* do robô virados para o exterior (figura 3.27a), dando um efeito especial do brilho e intensidade da cor (figura 3.27b).



(a) Localização dos LEDs no *chassis* do robô



(b) Colocação dos LEDs

Figura 3.27: Motores implementados no robô

O controlo dos LEDs RGB é realizado pela placa *Arduino*, através de sinais PWMs e independentes para cada cor vermelho, verde e azul. Apesar deste dispositivo possuir quinze pinos de saída PWM, este número é insuficiente para controlar todos os oito LEDs já que necessitamos de vinte e quatro PWMs para manipular cada cor.

A solução encontrada foi recorrer à programação do *Arduino* para criar PWMs suficientes pelos pinos de saída digitais. No desenvolvimento destes PWMs foram concebidos dez níveis de intensidade, que permitem obter qualquer cor com a combinação do código RGB.

Outra limitação encontrada na plataforma do *Arduino*, foi com a excessiva corrente do sistema de iluminação para a ativação de todos os LEDs. O *Arduino* está limitado a fornecer aos pinos de saída 200mA e esse valor é largamente ultrapassado, pois cada cor dos LEDs RGB necessita de

20mA. A solução encontrada para o total de 20mA*24=480mA foi utilizar os circuitos integrados *ULN2803*.

O esquema elétrico do sistema de LEDs é ilustrado na figura 3.28.

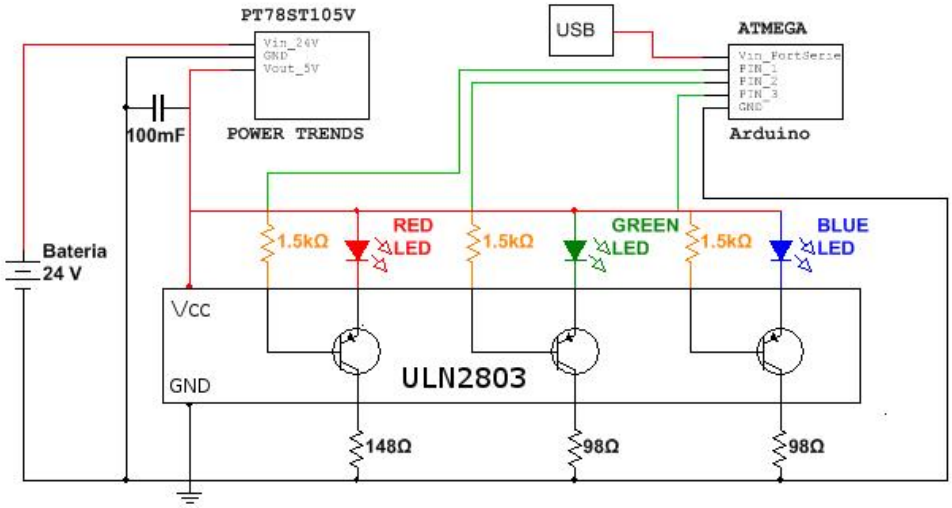


Figura 3.28: Esquema montagem do sistema de LEDs

Protocolo Comunicação com os LEDs

A comunicação entre ROS e o sistema de LEDs, realiza-se com a interface da placa *Arduino*. Os LEDs podem ser controlados ou por valores do código RGB específicos ou por funções com os mesmos valores já definidos.

A plataforma *Arduino* subscreve-se aos tópicos */Control_led*, no tipo de mensagem *std_msgs::UInt8MultiArray* e ao tópico */Mensagem* do tipo *std_msgs::String*.

O tópico */Control_led*, recebe um vetor de 24 valores inteiros que corresponde a cada cor dos LEDs. Foi desenvolvido uma interface para envio da trama de controlo dos 24 LEDs, podendo ser integrado em qualquer outro sistema do ROS (figura 3.29).

O tópico */Mensagem*, recebe no formato *string*, comandos enviados pelo sistema de controlo para realizar determinadas funções apresentadas na tabela 3.12.

Tabela 3.12: Funções dos comandos utilizados no sistema de leds (R - vermelho; G - verde; B - azul)

Descrição	Comando enviado	Nº dos leds ligados
Pisca Direito	RIGHT_BLINK	3 (RG), 4 (RG)
Pisca Esquerdo	LEFT_BLINK	7 (RG), 8 (RG)
Quatro Piscas	YELLOW_BLINK	3 (RG), 4 (RG), 7 (RG), 8 (RG)
Iluminação Frente (médios)	FRONT_LIGHT	5 (RGB), 6 (RGB)
Iluminação de Paragem	STOP_LIGHT	1 (R), 2 (R)
Sinalização de marcha trás	RETREAT_LIGHT	1 (RGB), 2 (RGB)

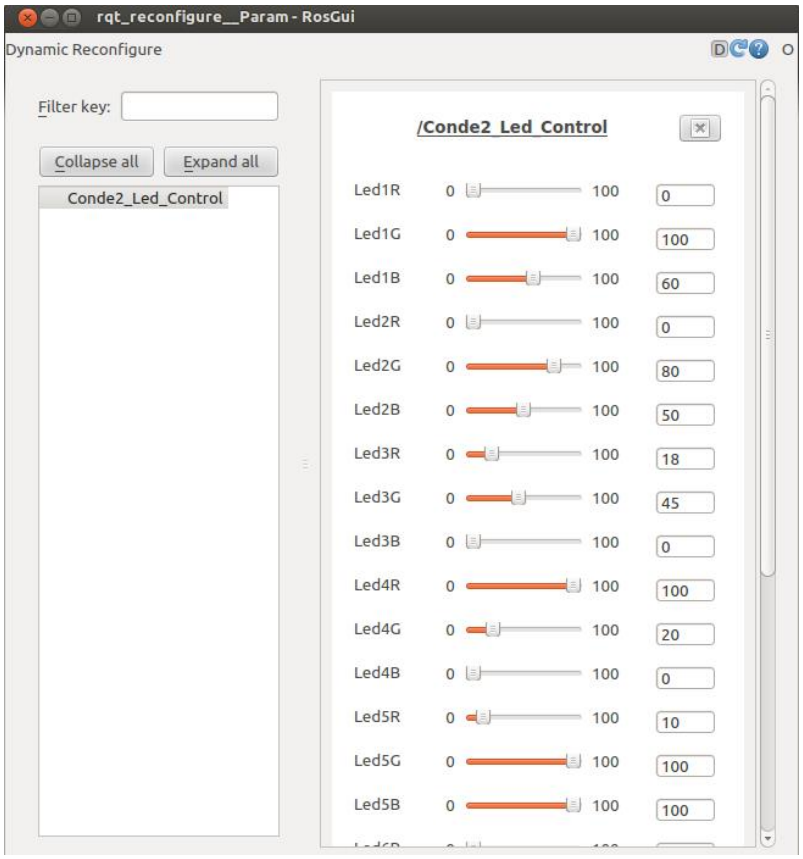


Figura 3.29: Interface desenvolvido para controlo independente de cada LED

Capítulo 4

Validação de Resultados

4.1 Comportamento motores

4.1.1 Calibração da Hodometria

Para validar o sistema de hodometria e a posição obtida pelos valores dos impulsos elétricos dos sensores de *Hall*, optou-se por realizar dois testes com percursos distintos: um em que o demonstrador percorre quatro metros em linha reta e outro em que o robô efetua uma rotação de 360º sobre si próprio. (figura 4.1)

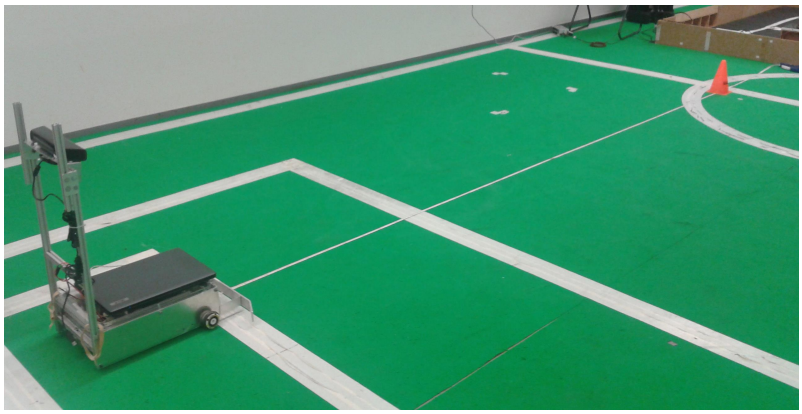
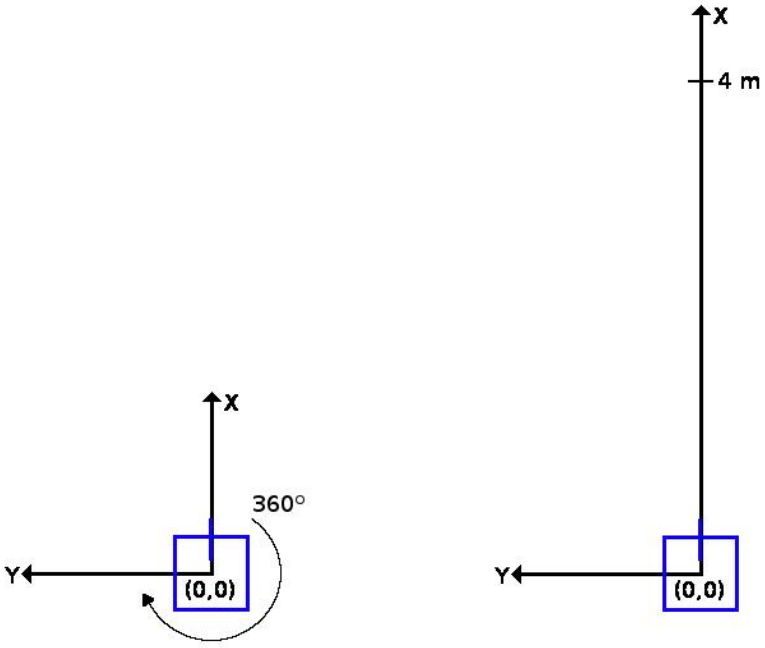


Figura 4.1: Cenário do teste de hodometria

A escolha destes testes satisfaz o objetivo pretendido, que se substância pela contagem dos impulsos elétricos com o mínimo de erro, já que as trajetórias são realizadas em movimentos simples, evitando os erros não sistemáticos, como a derrapagem ou o deslizamento das rodas (figura 4.2).

Apresenta-se na tabela 4.1 os valores médios dos impulsos elétricos obtidos no ensaio de quatro metros, para ambos os motores. Através destes dados obtém-se os valores para a roda esquerda de 1454 impulsos por metro e para a roda direita de 1441 impulsos por metro.



(a) Rotação do robô sobre si próprio

(b) Percurso segmento reta de quatro metros

Figura 4.2: Teste contagem dos impulsos elétricos

Tabela 4.1: Contagem em quatro metros dos Impulsos elétricos dos sensores de *Hall*

Linha reta	Motor Esquerdo	Motor Direito
1ª Medição	5811	5739
2ª Medição	5807	5759
3ª Medição	5799	5755
4ª Medição	5831	5769
5ª Medição	5807	5764
6ª Medição	5820	5791
7ª Medição	5809	5765
8ª Medição	5823	5757
9ª Medição	5825	5765
$\bar{x}$	5815	5763

No ensaio de rotação de 360º o demonstrador é manobrado de forma a dar uma volta completa em torno do seu centro de rotação. Foram recolhidas amostras para rotações no sentido dos ponteiros do relógio (PR) e contrário aos ponteiros do relógio (CPR). Através da comparação entre a hodometria e a medida real, procura-se neste teste obter o ponto central do eixo das rodas.

O efeito provocado pela incerteza no ponto de contacto das rodas, acontece apenas quando o robô efetua uma rotação. Por exemplo, num robô diferencial em que a expressão da velocidade angular do mesmo depende diretamente da distância entre eixos, se este não estiver bem calibrado, o robô irá rodar mais ou menos dependendo desse valor [7].

Para a obtenção do valor da distância entre os pontos de contacto das duas rodas ($dist_{axis}$), colocou-se o robô a rodar cinco vezes sobre si próprio. Recolhidos os valores finais dos impulsos (ΔIpM_n) de cada roda (tabela 4.2) e através das expressões (4.1) (4.2) verifica-se que o valor de $dist_{axis}$ é de 34.8 cm [7].

$$\Delta Dist_n = \frac{\Delta Counts_n}{\Delta IpM_n} \tag{4.1}$$

$$\Delta\theta = \frac{\Delta Dist_{left} - \Delta Dist_{right}}{dist_{axis}} \tag{4.2}$$

Em que $\Delta Dist_{left}$, $\Delta Dist_{right}$ são as distâncias percorridas, respetivamente, para a roda esquerda e direita e $\Delta\theta$ a variação da orientação do robô.

Tabela 4.2: Contagem dos Impulsos elétricos nos testes de rotação 360º

Rotação 360º	PR		CPR	
	Motor Esquerdo	Motor Direito	Motor Esquerdo	Motor Direito
1ª Medição	836	768	765	834
2ª Medição	826	772	830	769
3ª Medição	771	820	901	697
4ª Medição	774	823	928	682
5ª Medição	750	848	883	715
6ª Medição	795	795	925	688
$\bar{x}$	792	804	872	731

4.1.2 Calibração UMBMark

J.Borenstein e K.Feng [22], apresentaram um método sistemático de calibração do sistema de hodometria de um robô de tração diferencial, de forma a compensar o efeito dos erros sistemáticos.

O método de calibração *University of Michigan Benchmark test* (UMBmark) permite identificar separadamente o erro de diâmetros diferentes das rodas (E_d) e o erro da incerteza da distância entre os pontos de contacto das rodas com o chão (E_b). Desta forma este método permite compensar corretamente o efeito que cada um dos erros provoca na hodometria. Esta identificação é conseguida medindo a diferença entre a pose real e a pose medida pelo sistema de hodometria, após o demonstrador percorrer uma trajetória em forma de quadrado no sentido PR e repetir a mesma mas no sentido CPR.

Recorrendo às marcações realizadas no cenário do teste UMBmark, mediu-se o erro em X (Δx) e em Y (Δy) da hodometria no final do robô ter percorrido uma trajetória quadrangular com 3,20 m de lado (D)(figura 4.3). A trajetória foi efetuada a baixa velocidade (0.4m/s), de maneira a evitar a derrapagem das rodas. Foram realizados cinco ensaios tanto no sentido PR, como no sentido CPR, que se encontram apresentados na figura 4.4.

Observa-se que os resultados estão agrupados em duas regiões correspondentes ao sentido em que o percurso foi efetuado, evidenciando o efeito dos erros sistemáticos E_d e E_b. Estes resultados

Figura 4.3: Cenário do teste UMBmark

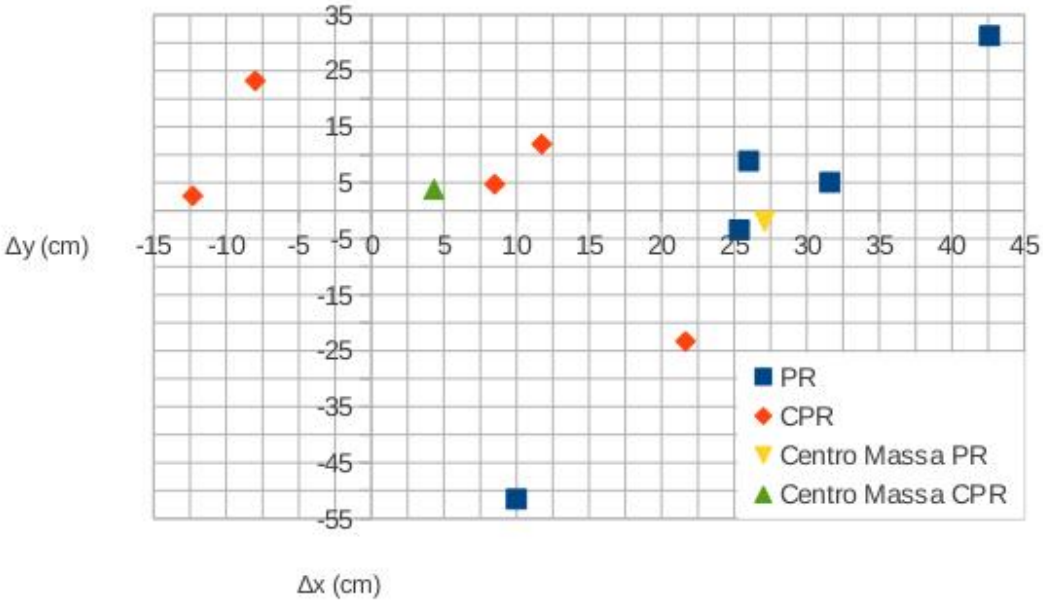


Figura 4.4: Erros da posição após o teste UMBmark

demonstram que o método permite isolar os efeitos de E_d e E_b, porque no sentido CPR da trajetória quadrangular estes efeitos compensam-se mutuamente (dando uma ideia errada da qualidade da hodometria). O mesmo não acontece quando efetuamos no sentido PR, onde os efeitos E_d e E_b somam-se um ao outro de forma a aumentar o erro total.

A partir destes dados, calcular-se o centro de massa do erro em X e em Y para cada região, demonstrados na tabela 4.3, através das expressões (4.3) (4.4) [23].

$$X_{CM} = \frac{1}{n}\sum_{i=1}^{n}\Delta x_i \tag{4.3}$$

$$Y_{CM} = \frac{1}{n}\sum_{i=1}^{n}\Delta y_i \tag{4.4}$$

Tabela 4.3: Centro de massa do teste UMBmark

	PR	CPR
X_{cm}	27,11	4.32
Y_{cm}	-1.93	3,83

O erro E_b afeta a hodometria apenas no movimento de rotação do robô, o que traduz num erro de medição da quantidade de rotação do robô (α). Já o erro E_d afeta apenas o movimento de translação do demonstrador, efeito que demonstra numa ligeira curvatura da trajetória ao longo dos quatro lados. A acumulação dos erros de orientação devido a esta curvatura é denominado por β.

Com os resultados de centro de massa calcula-se o ângulo β e o ângulo α pelas equações (4.5) (4.6), em que D é a dimensão do lado da trajetória quadrangular [22].

$$\alpha = média(\frac{X_{CM,PR} + X_{CM,CPR}}{-4 \times D}, \frac{Y_{CM,PR} - Y_{CM,CPR}}{-4 \times D}) \tag{4.5}$$

$$\beta = média(\frac{X_{CM,PR} - X_{CM,CPR}}{-4 \times D}, \frac{Y_{CM,PR} + Y_{CM,CPR}}{-4 \times D}) \tag{4.6}$$

Obteve-se α=-0.7935º e β=-0.5525º. Com o valor de α calcula-se o erro E_b (equação (4.7)) e sabendo que este erro é definido pela expressão (4.8) obtém-se a nova distância entre eixos b_{real}=34.49 cm.

$$E_b = \frac{90^\circ}{90^\circ - \alpha} \tag{4.7}$$

$$E_b = \frac{b_{real}}{b_{nominal}} \tag{4.8}$$

A partir de β calcula-se E_d com a expressão (4.9), onde $b_{nominal}$ é a distância entre eixos calculada anteriormente.

$$E_d = \frac{\frac{D/2}{sen(\beta/2)} + b/2}{\frac{D/2}{sen(\beta/2)} - b/2} \tag{4.9}$$

Com o cálculo de E_d obtém-se o novo diâmetro de cada roda da esquerda e da direita através das equações (4.10) (4.11), respetivamente, D_e=6,003 cm e D_d=5,997 cm. D_n corresponde ao valor do diâmetro nominal das rodas de 6 cm.

$$D_e = \frac{2}{E_d + 1} \times D_n \tag{4.10}$$

$$D_d = \frac{2}{\frac{1}{E_d} + 1} \times D_n \tag{4.11}$$

Com estes novos parâmetros voltou-se a repetir a experiência e obteve-se os resultados ilustrador na figura 4.5.

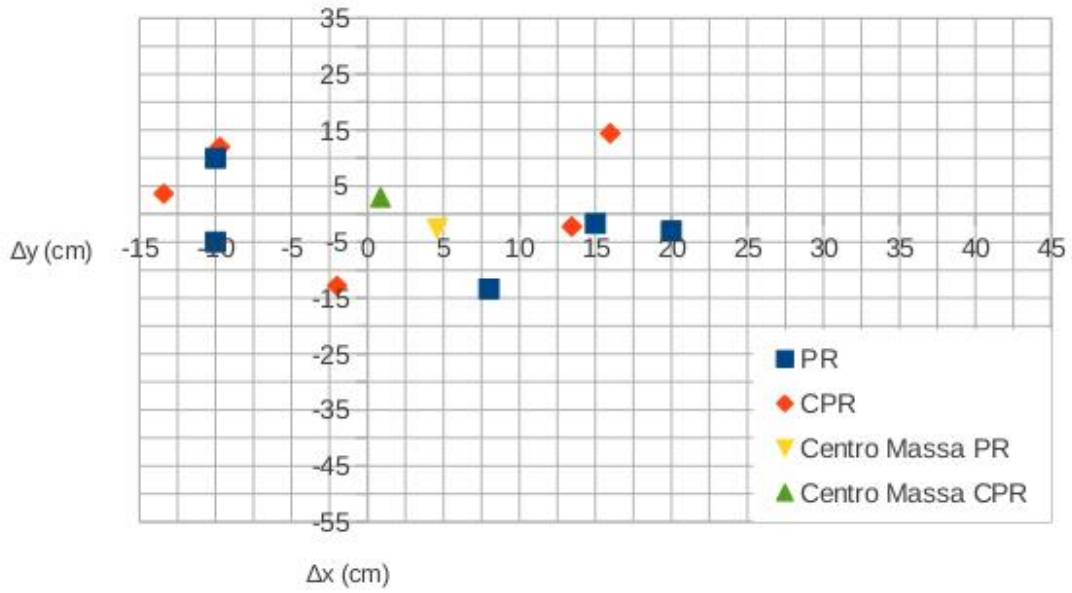


Figura 4.5: Erros da posição após o teste UMBmark com a nova calibração da hodometria

Como podemos constatar, já não se verificam dois conjuntos de pontos, o que indica uma boa compensação dos erros sistemáticos. Comparando a figura 4.4 e a figura 4.5 observa-se uma clara melhoria da precisão da hodometria.

4.2 Comportamento demonstrador

Nesta secção são apresentados testes de controlo que tencionam comprovar a eficácia dos controlos desenvolvidos, bem como ajustar parâmetros dos ganhos proporcionais. Foram realizados ensaios para dois tipos de controlos implementados: um seguimento de reta e uma trajetória similar ao teste UMBmark, onde se realizar em conjunto de seguimento de reta e ângulo.

Uma vez que nestes ensaios o objetivo é avaliar o desempenho do controlador, a precisão da hodometria não é discutida, sendo aceita como valor real.

4.2.1 Seguimento de reta

Este ensaio foi realizado de maneira análogo ao descrito na subsecção 4.1.1, entretanto, o seguimento de reta foi realizado pelo controlador, ao invés do utilizador.

Foram realizados ensaios de 3m, 5m e 8m, pretendendo avaliar a eficácia do controlador a curtas e longas distâncias, que são demonstradas nas figuras 4.6, 4.7 e 4.8.

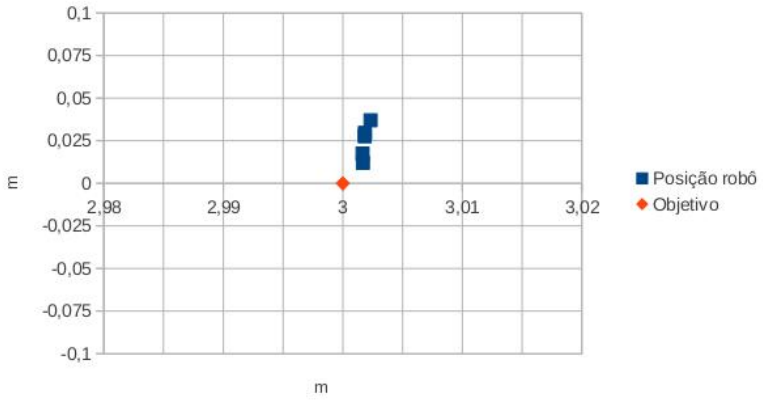


Figura 4.6: Resultado do teste de controlo numa linha reta de 3m

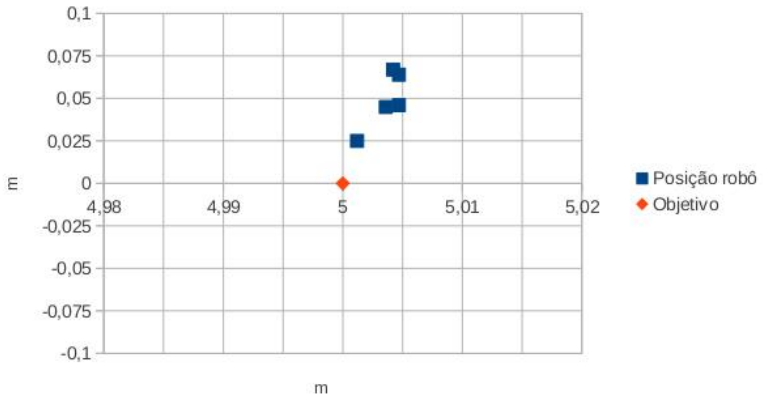


Figura 4.7: Resultado do teste de controlo numa linha reta de 5m

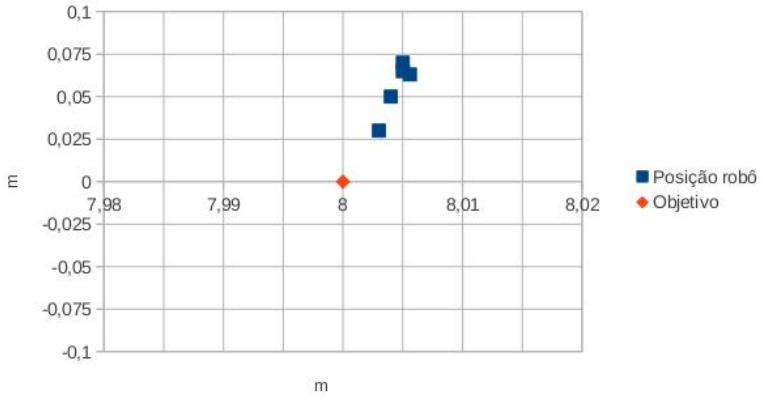


Figura 4.8: Resultado do teste de controlo numa linha reta de 8m

Os resultados analisados comprovam a eficácia do controlador implementado, que apresenta erros máximos de 8 cm para deslocamentos de grandes distâncias.

4.2.2 Seguimento de reta e ângulo

Neste ensaio realizou-se um percurso similar ao teste *UMBmark*, baseado numa trajetória quadrangular de 3.2m de lado. Com este teste pretende-se validar um só percurso o conjunto de seguimentos de reta e ângulo.

Foram realizados cinco testes no sentido CPR, que são representados na figura 4.9.

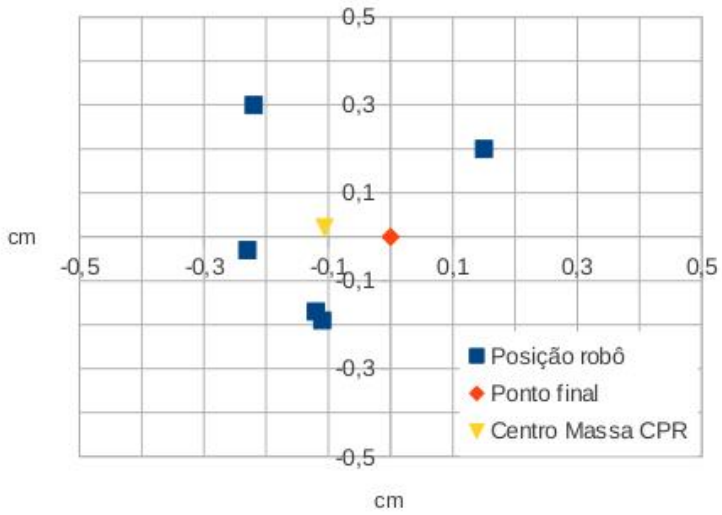


Figura 4.9: Resultado do teste de controlo para o percurso UMBmark no sentido CPR

Na realização destes ensaios utilizou-se os valores da hodometria publicados pelo pacote de controlo, no tópico */odom* (*nav_msgs::Odometry*) para a visualização no simulador *rviz*, como apresentado na subsecção 3.4.3 (figura 4.10).

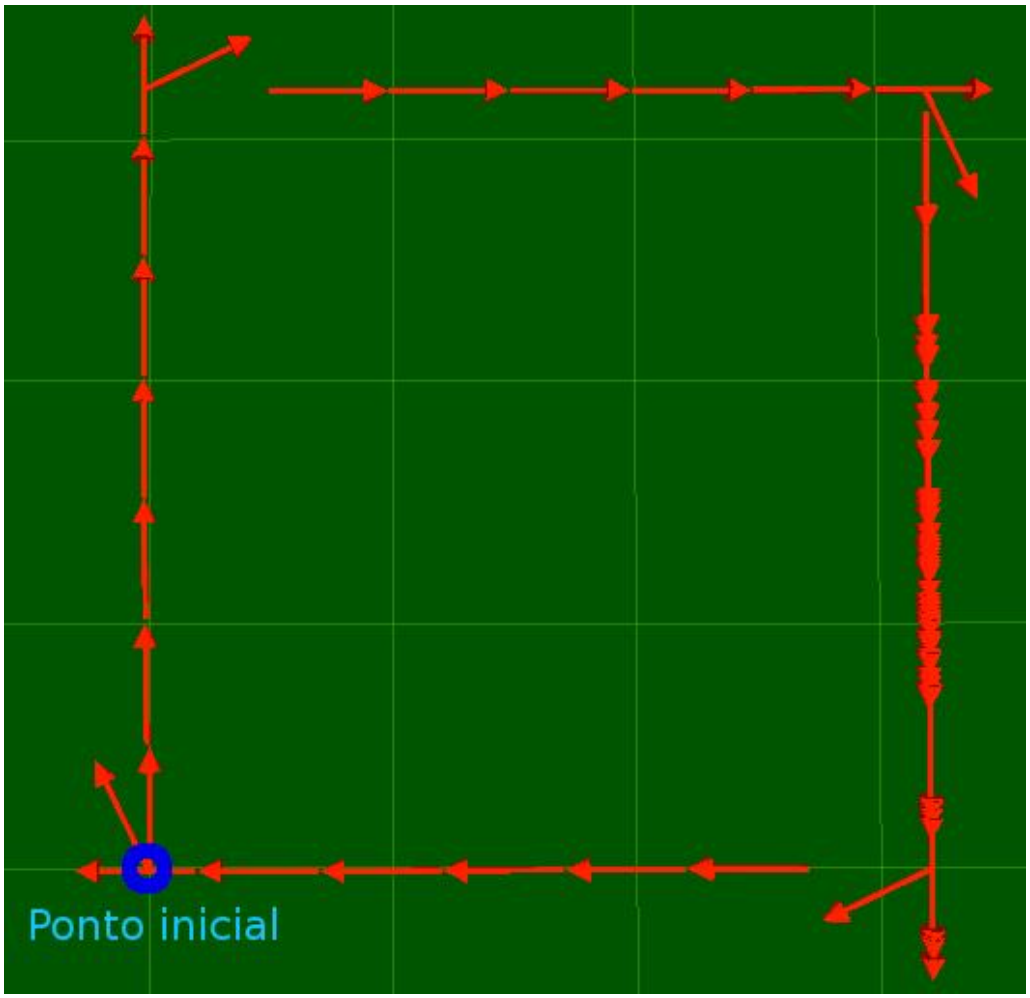


Figura 4.10: Visualização da hodometria durante o teste *UMBmark* no simulador *rviz*

Estes resultados apresentam o desempenho do controlador, para um percurso onde faz recurso do controlo do seguimento de reta e ângulo, com erros máximos de 3 cm. No simulador *rviz* podemos observar que o controlo do seguimento de reta foi mais intenso no segmento do lado direito do percurso, onde podemos verificar que cada valor da hodometria (seta vermelha), indica a posição (ponto inicial da seta) e a orientação do robô (direção da seta).

Capítulo 5

Conclusão

Nesta dissertação estudaram-se e implementaram-se diferentes funcionalidades e aplicações da plataforma ROS, direcionadas para o sistema de locomoção e controlo. Do estudo realizado verificou-se que o leque de soluções disponibilizadas, o fato de ser livre, de código aberto, a sua eficiência quanto à comunicação e coordenação de mensagens, serviços e a sua flexibilidade tornam o ROS apto a ser um sistema líder, com desafios em grande escala. Exemplos como o robô PR2 mostram o quanto este *framework* é revolucionário e com grande potencial, quanto ao desenvolvimento de aplicações focadas na área da robótica.

Realizou-se a implementação de novos motores, que obrigou a uma nova proposta da arquitetura de *hardware* do demonstrador. Esta nova arquitetura solucionou a necessidade de controlo do motor por comandos PWM provenientes da placa eletrónica *Arduino*.

Foi estudado o protocolo de comunicação e desenvolvido um sistema de comunicação por porta de série RS232 com as *drivers* do robô, aumentando a capacidade de processamento da plataforma *Arduino*, que se encontrava a realizar esta operação. Foi ainda desenvolvida uma aplicação robusta, segura na plataforma do ROS, que realiza a comunicação com as *drivers*, para envio de comandos de velocidade, configurações e testes. Verificou-se que as *drivers dzr* possibilitam o *feedback* dos sensores dos impulsos do motor pela porta séria, apenas para os *encoders*, estando limitada para os sensores *hall*.

Estudou-se e aplicou-se uma técnica de calibração da hodometria, que permite uma correção eficiente dos erros sistemáticos, melhorando significativamente a posição do robô. Conclui-se que as pequenas variações dos parâmetros do sistema de hodometria, como o diâmetro entre eixos e o diâmetros das rodas, são especialmente críticos, para uma localização relativa nos robôs diferenciais. Apesar de os resultados serem bastantes satisfatórios chegou-se à conclusão que um sistema extra de localização seria um valor acrescentado para a eliminação dos erros acumulativos, que o sistema de hodometria acarreta.

Desenvolveu-se um *sketch*, com ligação ao *package rosserial*, na plataforma *Arduino* capaz de realizar uma contagem segura e sem falhas dos impulsos elétricos provenientes dos sensores *hall*, bem como foi possível estabelecer uma comunicação com o protocolo do sistema de LEDs. Verificou-se que a sincronização entre *Arduino* e a plataforma ROS deve ser bem estruturada para

que não se perca informação essencial, devido à falha da comunicação entre as duas plataformas.

Implementou-se um sistema de LEDs e o seu respetivo protocolo de comunicação em ROS, capaz de funcionar como uma excelente ferramenta de teste, como de realizar operações de demonstração associadas aos movimentos do robô.

Sobre o sistema de controlo foram desenvolvidos seguimentos de trajetória, concretamente, seguimento por reta, ponto e ângulo, que permitem realizar correções na locomoção do robô. Os ensaios do controlo foram eficazes para a sua validação e importantes para o ajuste dos parâmetros dos ganhos do controlo. Conclui-se que, os seguimentos de trajetórias são uma das possíveis soluções para o controlo do robô, sendo factível representar qualquer combinação de percursos pela simplificação de movimentos básicos.

Foram ainda desenvolvidas diversas aplicações na plataforma ROS, como um modelo *urdf* do robô, que permite a visualização da estrutura do mesmo numa plataforma de simulação, um *package* que permite o controlo remoto do demonstrador pelo teclado ou comando *Wii*, um sistema de deteção de sinais e semáforos utilizados na competição nacional de robótica de condução autónoma, que podem ser implementadas e utilizadas em futuros projetos relacionados com a plataforma ROS.

5.1 Futuros desenvolvimentos

Como trabalhos futuros pretende-se realizar um porte do sistema de visão para a plataforma ROS, permitindo assim criar um sistema de localização absoluto, como por exemplo deteção de balizas e gps.

Ainda no sistema de visão e devido ao variado leque de funcionalidades e serviços estudados da plataforma ROS, surgiram várias soluções para a aplicação de um sistema de navegação de localização e mapeamento simultâneos (SLAM) utilizando os *frames tf* desenvolvidos, juntamente com o sistema de hodometria e com o sensor *kinect*.

Outro aspeto relevante nesta dissertacão seria o desenvolvimento de controladores do tipo PID, que permitisse ao demonstrador melhorar o desempenho no campo de planeamento de trajetórias. Igualmente importante seria a finalização do trabalho de comunicação com o simulador, abrindo novas possibilidade de navegação em ambientes virtuais e de caraterísticas diferentes às apresentadas neste estudo.

Anexo A

Índices dos comando da *driver* AMC

Lista dos índices em hexadecimal dos comandos básicos da *driver* AMC a utilizar no *Index Byte*.

Tabela A.1: Lista do *byte index*

INDEX	OPERATION	INDEX	OPERATION
01h	Control Parameters	36h	Velocity Loop Control Parameters
02h	Drive Status	37h	Velocity Limits
03h	Drive Status History	38h	Position Loop Control Parameters
04h	Heartbeat Parameters	39h	Position Limits
05h	Serial Interface Configuration	3Ah	Homing Configuration Parameters
06h	Network Configuration	3Ch	Command Profiler Parameters
07h	Access Control	3Dh	Deadband Parameters
08h	PVT Quick Status	3Eh	Jog Parameters
09h	Restore Drive Parameters	43h	Capture Configuration Parameters
0Ah	Store Drive Parameters	44h	Analog Input Parameters
0Bh	Stored User Parameters	45h	Interface Inputs
0Eh	Feedback Sensor Values	46h	Auxiliary Input Parameters
0Fh	Power Bridge Values	48h	PVT Parameters
10h	Current Values	54h	Drive Temperature Parameters
11h	Velocity Values	58h	Digital Input Parameters
12h	Position Values	5Ah	Digital Output Parameters
14h	Command Limiter Input	5Ch	Analog Output Parameters
15h	Deadband Input	62h	Braking/Stop General Properties
19h	Capture Values	64h	Fault Response Time Parameters
1Ah	Analog Input Values	65h	Fault Event Action Parameters
1Ch	Auxiliary Input Values	66h	Fault Recovery Time Parameters
1Dh	PVT Status Values	67h	Fault Time-Out Window Parameters

INDEX	OPERATION	INDEX	OPERATION
21h	Drive Temperature Values	68h	Fault Maximum Recoveries Parameters
23h	Digital Input Values	8Ch	Product Information
24h	Digital Output Values	8Dh	Firmware Information
25h	Analog Output Values	C8h	Motion Engine Configuration
27h	Feedback Hardware Diagnostics	C9h	Motion Engine Control
28h	Fault Log Counter	D0h	Control Loop Configuration Parameters
29h	Motion Engine Status	D1h	Mode Configuration
32h	Feedback Sensor Parameters	D3h	Active Mode Configuration
33h	User Voltage Protection Parameters	D8h	Power Board Information
34h	Current Loop & Commutation Control Parameters		

Anexo B

Tabela CRC

Neste anexo encontra-se a tabela CRC de verificação e validação da trama de comunicação com a *driver*.

0000	1021	2042	3063	4084	50A5	60C6	70E7
8108	9129	A14A	B16B	C18C	D1AD	E1CE	F1EF
1231	0210	3273	2252	52B5	4294	72F7	62D6
9339	8318	B37B	A35A	D3BD	C39C	F3FF	E3DE
2462	3443	0420	1401	64E6	74C7	44A4	5485
A56A	B54B	8528	9509	E5EE	F5CF	C5AC	D58D
3653	2672	1611	0630	76D7	66F6	5695	46B4
B75B	A77A	9719	8738	F7DF	E7FE	D79D	C7BC
48C4	58E5	6886	78A7	0840	1861	2802	3823
C9CC	D9ED	E98E	F9AF	8948	9969	A90A	B92B
5AF5	4AD4	7AB7	6A96	1A71	0A50	3A33	2A12
DBFD	CBDC	FBBF	EB9E	9B79	8B58	BB3B	AB1A
6CA6	7C87	4CE4	5CC5	2C22	3C03	0C60	1C41
EDAE	FD8F	CDEC	DDCD	AD2A	BD0B	8D68	9D49
7E97	6EB6	5ED5	4EF4	3E13	2E32	1E51	0E70
FF9F	EFBE	DFDD	CFFC	BF1B	AF3A	9F59	8F78
9188	81A9	B1CA	A1EB	D10C	C12D	F14E	E16F
1080	00A1	30C2	20E3	5004	4025	7046	6067
83B9	9398	A3FB	B3DA	C33D	D31C	E37F	F35E
02B1	1290	22F3	32D2	4235	5214	6277	7256
B5EA	A5CB	95A8	8589	F56E	E54F	D52C	C50D
34E2	24C3	14A0	0481	7466	6447	5424	4405
A7DB	B7FA	8799	97B8	E75F	F77E	C71D	D73C
26D3	36F2	0691	16B0	6657	7676	4615	5634
D94C	C96D	F90E	E92F	99C8	89E9	B98A	A9AB
5844	4865	7806	6827	18C0	08E1	3882	28A3
CB7D	DB5C	EB3F	FB1E	8BF9	9BD8	ABBB	BB9A
4A75	5A54	6A37	7A16	0AF1	1AD0	2AB3	3A92
FD2E	ED0F	DD6C	CD4D	BDAA	AD8B	9DE8	8DC9
7C26	6C07	5C64	4C45	3CA2	2C83	1CE0	0CC1
EF1F	FF3E	CF5D	DF7C	AF9B	BFBA	8FD9	9FF8
6E17	7E36	4E55	5E74	2E93	3EB2	0ED1	1EF0

Figura B.1: Tabela CRC

A tabela pode ser criada com recurso às bibliotecas do IDE *Qtcreator* e na linguagem C++ utilizando o seguinte código.

```
#include <QObject>
#include <QVector>
#include <QByteArray>

#define CRC_POLY 0x1021

QVector<u_short> FCRCTable;

void MakeCRCTable(QVector<uint16_t> &CRCTable)
{
  CRCTable.resize(256);
  for(uint16_t i=0; i<256; ++i){
    CRCTable[i]=crchware(i, CRC_POLY, 0);
  }
}

uint16_t crchware(uint16_t data, uint16_t genpoly, uint16_t accum)
{
  data<<=8;
  for(int i=8; i>0; i--){
    if((data ^ accum) & 0x8000){
      accum = (accum << 1 ) ^ genpoly;
    }
    else{
      accum <<=1;
    }
      data <<=1;
  }
  return accum;
}
```

Anexo C

ROS no QTCreator

Abrir projeto (*package*) no Qt

1) Correu Qt através da consola.

```
$ qtcreator
```

2) "*Open Project*" e seleccionar *CMakeList.txt* do *package* que pretendemos.

Dica importante: Arrancar Qt via terminal, caso contrário Qt não localiza bibliotecas do ROS, impedindo conclusão *CMake*.

Trabalhar com bibliotecas do Qt em ROS

1) Adicionar ao ficheiro *CmakeList.txt* os seguintes códigos:

```
FIND_PACKAGE(Qt4 REQUIRED)
ADD_DEFINITIONS(${QT_DEFINITIONS})
ADD_DEFINITIONS(-DQT_NO_KEYWORDS)
SET(QT_USE_QTNETWORK TRUE)

QT4_WRAP_CPP(PROGRAM_HEADERS_MOC ${PROGRAM_HEADERS})
INCLUDE(${QT_USE_FILE})
```

2) Adicionar no *TARGET_LINK_LIBRARIES*, acesso às bibliotecas do *QtCreator*:

```
TARGET_LINK_LIBRARIES(nome_node ${QT_LIBRARIES})
```

Anexo D

Porte C++ para CMakeList ROS

Primeiro passo consiste na criação do *package* na área de trabalho do ROS. Na criação deverá indicar que dependências irá utilizar. Poderá adiciona-las mais tarde com alteração do *manifest.xml*.

Sintaxe do comando:

roscreate-pkg [package_name] [depend1] [depend2] [depend3]

Exemplo:

```
roscreate-pkg cam_semaforo roscpp std_msgs sensor_msgs opencv2
```

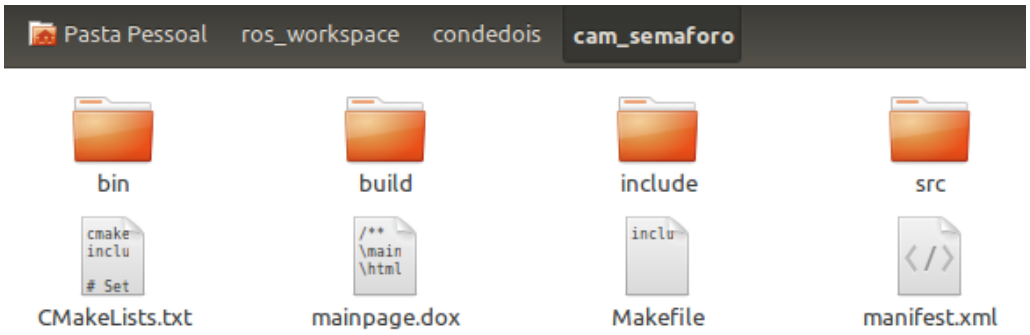


Figura D.1: Visualização da pasta criada no exemplo

Changes in CmakeList.txt:

rosbuild_init() : invokes rospack to retrieve the build flags for this package.

rosbuild_add_executable : call says that you want to build an executable called *driver_dzr_velManual* from the sources files in the set PROGRAM_SOURCES.

target_link_libraries() : to link against an external library (Qt_Libraries, OpenCV, ...)

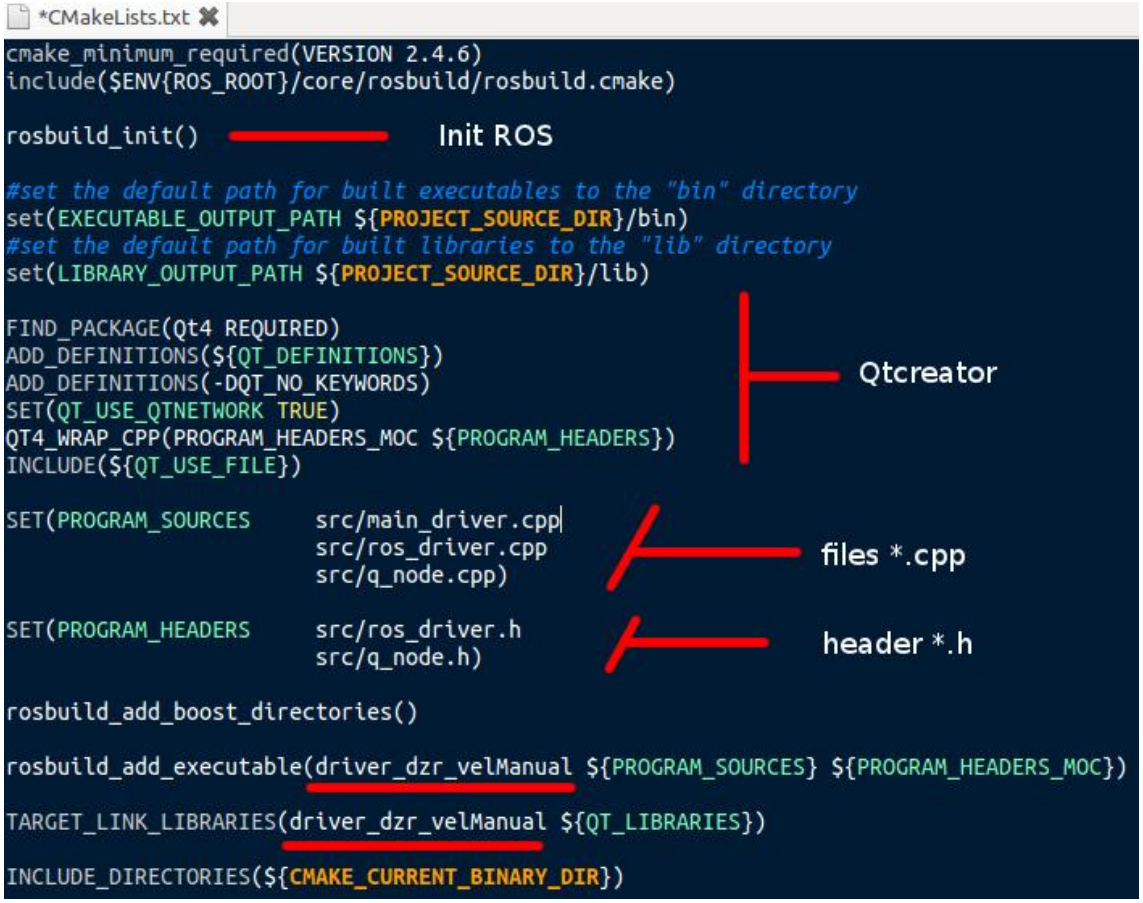


Figura D.2: Estrutura do ficheiro CmakeList.txt

Files *.cpp

1) ros::init(argc,argv,"name_node")

Initializing the node through a call to one of the *ros::init()* functions. This provides command line arguments to ROS, and allows you to name your node and specify other options.

2) ros::NodeHandle n

Startup and shutdown of the internal node inside a roscpp program. Once all *ros::NodeHandle* instances have been destroyed, the node will be automatically shutdown.

3) subscribe("/topic_name", size, callback)

/topic_name: The topic to subscribe to.

size: incoming message queue size.

<*callback*>: piece of executable code that is passed as an argument to other code.

4) advertise<data_type>("/topic_name", size)

/topic_name: This is the topic to publish on.

size: size of the outgoing message queue.

5) spinOnce()

specify a threading model for your application. your subscription, service and other callbacks will be called with solution *ros::spin()*.

Others) ros::master::check()

Check on the status of the master. (roscore)

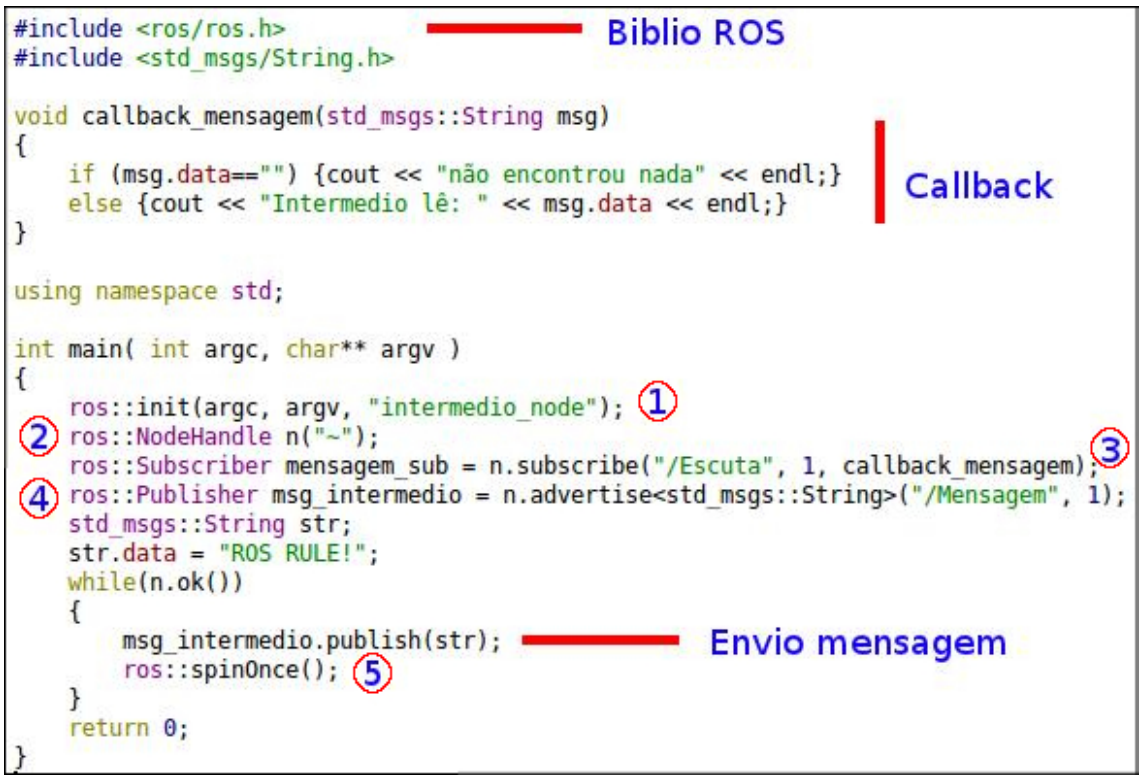


Figura D.3: Estrutura de um ficheiro *.cpp

Anexo E

Modelo Demonstrador - URDF

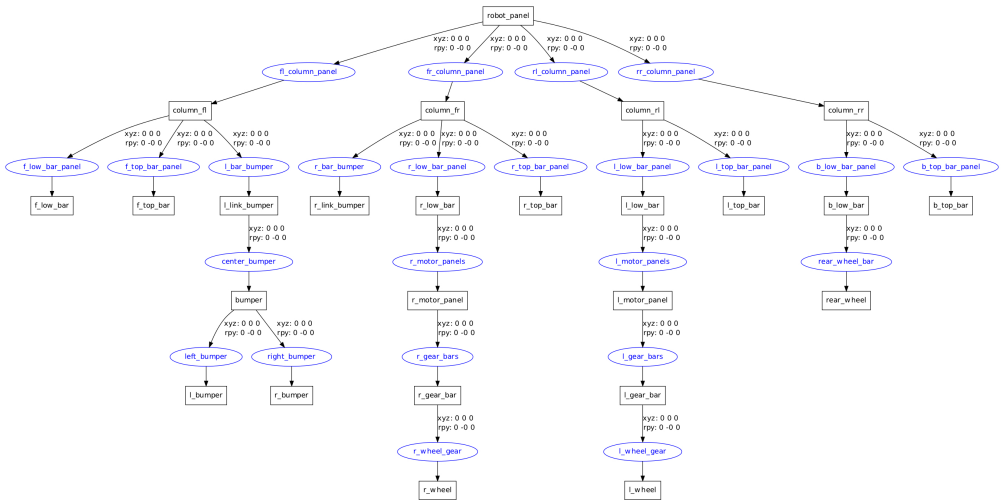


Figura E.1: Representação do modelo do robô no formato *URDF*

Referências

[1] *Robotics Operating System*. Disponível em http://www.ros.org/wiki/, acedido a última vez em 14/02/2013.

[2] Urbi. Disponível em http://www.gostai.com/, acedido a última vez em 15/02/2013.

[3] *Microsoft robotics*. Disponível em http://www.microsoft.com/robotics/, acedido a última vez em 15/02/2013.

[4] Dave. Disponível em http://dros.org/, acedido a última vez em 12/02/2013.

[5] *iRobot Create*. Disponível em http://www.irobot.com/create/, acedido a última vez em 12/02/2013.

[6] Pierre Blazevic Chris Kilner Jeróme Monceaux Pascal Lafourcade Brice Marnier Julien Serre Bruno Maisonnier David Gouaillier, Vincent Hugel. "mechatronic design of nao humanoid". *IEEE JOURNALS*, Maio 2009.

[7] José Júlio Areal Ferreira. "demonstrador de condução autónoma", Fevereiro 2012.

[8] Arduino mega 2560. Disponível em http://arduino.cc/en/Main/arduinoBoardMega2560, acedido a última vez em 17/05/2013.

[9] Advanced Motion Controls. "serial communication - reference manual", Julho 2012. Disponível em http://www.a-m-c.com/support/downloadable_resources.html, acedido a última vez em 25/06/2013.

[10] Surachai Panich e Nitin Afzulpurkar. "sensor fusion techniques in navigation application for mobile robot", Junho 2011. Capítulo 6.

[11] *Robotics Operating System*. Adaptado de http://www.ros.org/wiki/, acedido a última vez em 14/02/2013.

[12] Advanced Motion Controls. "serial communication - reference manual", Julho 2012. Adaptado em http://www.a-m-c.com/support/downloadable_resources.html, acedido a última vez em 25/06/2013.

[13] Steve Cousins. "exponential growth of ros". *IEEE JOURNALS*, Março 2011.

[14] Ken Conley Josh Faust Tully Foote Jeremy Leibs Eric Berger Rob Wheeler Morgan Quigley, Brian Gerkey e Andrew Ng. "ros: an open-source robot operation system". 2009.

[15] Brian Gerkey Steve Cousins e Willow Garage. "sharing software with ros". *IEEE JOURNALS*, Junho 2010.

[16] *MotionShop*. Disponível em http://www.motionshop.com/, acedido a última vez em 20/06/2013.

[17] Advanced Motion Controls. "driveware 7.0 software manual", Julho 2012. Disponível em http://www.a-m-c.com/support/downloadable_resources.html, acedido a última vez em 25/06/2013.

[18] L. Tosini F. Ribeiro e G. Lopes. "localization of a mobile autonomous robot based on image analysis", 2007.

[19] Stergios I. Roumeliotis Puneet Goel e Gaurav S. Sukhatme, Outubro 1999.

[20] Roland Siegwart e Illah R. Nourbakhsh. "introduction to autonomous mobile robots", 2004.

[21] Marcelo Roberto Petry. "desenvolvimento de um protótipo e de metodologias de controlo de uma cadeira de rodas inteligente", Fevereiro 2008.

[22] J. Borenstein e L. Feng. "umbmark - a method for measuring, comparing, and correct- ing dead-reckoning errors in mobile robots", Dezembro 1994.

[23] Héber Miguel Plácido Sobreira. "clever robot", Julho 2009.